Policy Conflict Analysis for Quality of Service Management

Marinos Charalambides[1], Paris Flegkas[1], George Pavlou[1], Arosha K Bandara[2], Emil C Lupu[2], Alessandra Russo[2], Naranker Dulay[2], Morris Sloman[2], Javier Rubio-Loyola [3]
[1]*University of Surrey,* [2]*Imperial College London,* [3]*Universitat Politècnica de Catalunya*
[1]*{M.Charalambides, P.Flegkas, G.Pavlou}@eim.surrey.ac.uk,* [2]*{a.k.bandara, e.c.lupu, a.russo, n.dulay, m.sloman}@imperial.ac.uk,* [3]*jrloyola@tsc.upc.edu*

Abstract

Policy-based management provides the ability to (re-)configure differentiated services networks so that desired Quality of Service (QoS) goals are achieved. Relevant configuration involves implementing network provisioning decisions, performing admission control, and adapting bandwidth allocation dynamically according to emerging traffic demands. A policy-based approach facilitates flexibility and adaptability in that the policies can be changed without changing the implementation. However, as with any other complex system, conflicts and inconsistencies may arise in the policy specification. In this work, we concentrate on the policy conflicts that may occur for static resource management aspects of QoS provisioning, known as Network Dimensioning. The paper shows how conflict detection can be achieved using Event Calculus in conjunction with abductive reasoning techniques to detect the existence of potential conflicts in partial specification and generate explanations for the conditions under which the conflicts arise. We finally present some conflict detection examples from our initial implementation of a policy conflict analysis tool. Although we focus on network dimensioning, many of the types of conflicts we illustrate could arise in other applications.

1. Introduction

In recent years, fully-automated, policy-based management has been proposed as a suitable means for managing Quality of Service (QoS) in IP networks. Yet despite research projects, standardisation efforts, and substantial interest from industry, policy-based management is still not a reality. There are some vendor tools, mostly part of virtual private network provisioning toolsets, but policy-based management is still far from being widely adopted despite its potential benefits of flexibility and "constrained programmability". One of the reasons behind the reticence to adopt this technology is that it is difficult to analyse policies in order to guarantee network configuration stability given that policies may have conflicts leading to unpredictable effects. There are no policy analysis tools that can detect policy conflicts beyond simple cases or identify the circumstances in which a conflict may arise.

Initial work on policy analysis focused on identifying modality conflicts addressing simple analysis between positive and negative authorisation security policies and the specification of policy precedence rules in order to resolve conflicts [1]. In addition, Jajodia in [2] has proposed a logic-based specification of security policies with relatively simple well-understood semantics amenable to analysis, using techniques based on deductive reasoning; these are in general not suitable for reasoning over incomplete specifications or for identifying causes of conflicts. The work in this paper is based on the work presented in [3] where the use of Event Calculus (EC) was proposed as a specialised first-order logic for formalising policy specification based on solid theoretical foundations [4] and the mapping to and from the Ponder policy language [5]. EC allows specification of the system behaviour using familiar notations, such as state charts, which can then be automatically translated into the logic program representation. Abductive reasoning proof procedures for EC [6] can be used to detect the existence of potential conflicts in partial specifications and generate explanations for the conditions under which such conflicts may arise.

The initial work mentioned above showed how the EC formalism can be used in conjunction with abductive reasoning techniques to perform a priori analysis of policy specifications for the generic conflict types presented in the literature. While this work proposes a promising methodology to tackle the problem of conflict analysis in a generic fashion, it is not sufficient to provide a complete solution to the

problem without addressing the needs of an application-specific domain. In this paper, we extend and refine the aforementioned approach of using EC for conflict analysis by applying it to the domain of QoS Management of IP Differentiated Services (DiffServ) Networks. In order to identify the policies and conflicts involved in DiffServ QoS management, we use the framework developed in the context of the EU IST TEQUILA project [7]. TEQUILA uses DiffServ together with Multi-Protocol Labelled Switching (MPLS) to support a network that can dynamically adapt to varying network traffic demands. More specifically, we focus on conflicts that may arise from policies driving the Network Dimensioning (ND) component of the TEQUILA framework. ND is part of the resource management sub-system and is responsible for mapping traffic forecast requirements onto the physical network resources by providing configuration directives in order to accommodate the predicted traffic demand. Relevant policies and associated enforcement examples have been presented in previous work [8].

We identify and provide a taxonomy of the different conflicts that can emerge from policy specifications that drive the behaviour of ND and show how the EC formalism can be used to support the specification of rules for detecting different conflict types. Using abductive reasoning, we are able to analyse the policy specifications to identify existing conflicts and provide explanations on how they might arise. The latter is demonstrated by conflict detection examples taken from our initial implementation of a conflict analysis tool. The work builds upon the initial approach described in [3]. The paper focuses on analysing application-specific conflicts which has hardly been addressed in the literature for management policies. However many of the types of conflicts we discuss, such as mutual exclusion or resource allocation also arise in other application areas and so the concepts described could be easily adapted to any application.

In the next section we present some background information about EC and policy analysis, as well as a short description of the TEQUILA framework focusing on the behaviour of the ND component. Section 3 details the identified policies for Network Dimensioning along with their representation in the Ponder specification language. In section 4 we present the classification of the identified conflict types as well as the conditions under which these conflicts may arise. Section 5 presents the rules for detecting the conflicts along with specific conflict detection examples. Finally section 6 presents some related work in this field; and section 7 discusses our conclusions and future work.

2. Background

2.1. Formal representation and Event Calculus

Event Calculus is a formal language 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. Since its initial presentation [4], a number of variations have been presented in the literature. In this work we use the form presented in [9], 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`, `terminates`, `holdsAt`, `happens`, as summarised below:

Base predicates:	
initiates(A,B,T)	event A initiates fluent B for all time > T.
terminates(A,B,T)	event A terminates fluent B for all time > T.
happens(A,T)	event A happens at time point T.
holdsAt(B,T)	fluent B holds at time point T.
initiallyTrue(B)	fluent B is initially true.
initiallyFalse(B)	fluent B is initially false.

This is the classical form of the 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 `holdsAt`, `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.

Event Calculus supports deductive, inductive and abductive reasoning. The technique that is of particular interest to our work is abduction. Given the descriptions of the behaviour of the system, abduction can be used to determine the sequence of events that need to occur so that a given set of fluents will hold at a specified point in time.

Table 1: Function symbols.

Symbol
operation(Obj, Action(V_P))
doAction(Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)))
SystemEvent(Event)
permit(PolID,Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))
oblig(PolID, Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))
refrain(PolID,Obj_{Subj},operation(Obj_{Targ}, Action (V_P)))

The work described in [9] outlines how abduction can be used in conjunction with Event Calculus to analyse requirements specifications and presents a

specialised set of EC axioms that reduce the computational complexity of the abductive proof procedure. The formal language being used is based on that described in [2], where in addition to the base predicates and axioms of Event Calculus we make use of the function symbols shown in Table 1.

2.2. Policy analysis

In an environment where a number of policies need to coexist, there is always the likelihood that several policies will be in conflict, either because of a specification error or because of application-specific constraints. It is therefore important to provide a means of detecting conflicts in the policy specification.

The different types of conflicts that can occur are identified in [1]. Modality conflicts arise when two policies are specified using the same subjects, targets and actions but are of opposite modality (e.g. obligation and refrain). This type of conflict is domain-independent since conflicts could occur irrespective of the application domain for which the policies are being specified. Other conflict types identified in the literature fall into the category of application-specific conflicts. As described in [10], these include conflicts of duty, conflicts of interest, multiple manager conflicts, conflicts of priorities for resources and self-management conflicts.

Considering the types of conflicts described above, it is possible to define rules that can be used to recognise conflicting situations in the policy specification. Modality conflicts involving obligation and refrain policies occur when the two policies are defined for the same subject, target and action. The `obligConflict` predicate defined below holds if a modality conflict is detected.

```
holdsAt(obligConflict(Subj, Op), T) ←
    holdsAt(oblig(Subj, Op), T) ∧
    holdsAt(refrain(Subj, Op),T).
```

In the above rule, the `Op` variable will be instantiated with an `operation` term as defined in Table 1.

One of the most common types of application-specific conflict cited in the literature is conflict of duties (alternatively stated as the requirement to ensure separation of duties) [2, 10]. A conflict of duties will arise if the same subject is permitted to perform operations that, in the context of the application, are defined to be conflicting. For example, in a company financial system, the operation of entering a request for payment and the operation of approving that request are potentially conflicting if the same user can perform both operations.

Rules for application-specific conflicts must be defined using constraints that include application-specific data in addition to policy information. In order to capture the additional information, we extend the system specification language to include rules that define each application-specific conflict that may arise in the system. The rules can include ground literals, specifying the action/target object combinations that will potentially conflict. In the case of the conflict of duties example mentioned above, the potential conflict between the operations of requesting a payment and approving a payment would be represented as follows:

```
holdsAt(sepOfDutyConflict(Subj, Ops), T) ←
    holdsAt(permit(Subj, operation(payment,
            request(PaymentID, Amount))), T1) ∧
    holdsAt(permit(Subj, operation(payment,
            approve(PaymentID))), T2) ∧
    T1 =< T2.
```

Similar rules can be specified for other types of application specific conflicts, such as conflicts of interest and self-management conflicts [3].

2.3. QoS management

Management plane functionality [11] is needed to support end-to-end quality of service based on Service Level Subscriptions (SLSs), using DiffServ Per Hop Behaviour (PHB) together with Multi-Protocol Labelled Switching (MPLS). A policy-based functional architecture for supporting quality of service in IP DiffServ Networks has been designed in the context of the European collaborative research project TEQUILA (Traffic Engineering for QUality of service in the Internet at LArge scale). This architecture can be seen as a detailed decomposition of the concept of a Bandwidth Broker realized as a hierarchical, logically and physically distributed system and has been presented in [7]. It is decomposed into three major sub-systems: SLS management, Traffic Engineering 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. We describe below the functionality of the Network Dimensioning component which is part of the Traffic Engineering sub-system since the rest of the paper focuses on the analysis of the policies driving its behaviour.

Network Dimensioning performs the provisioning activities of the management system. It is responsible for the long to medium term configuration of the network. By configuration we mean the setup of Label Switched Paths (LSPs) as well as the parameters (e.g. priority, weight, bandwidth) required for the operation of PHBs on every link. The values provided by ND are not absolute but come in the form of a range, constituting directives for the function of the PHBs,

while for label switched paths they come in the form of multiple paths in order to enable multi-path load balancing. The exact configuration values and the chosen path among the multiple paths to be used are determined by dynamic TE functions based on the actual state of the network at any point in time.

ND runs periodically by first requesting the predictions for the expected traffic per traffic class or Ordered Aggregate (OA), i.e. EF, AF1x, AF2x, AF3x or BE for DiffServ, in order to be able to compute the provisioning directives. The dimensioning period is typically in the time scale of a week and the goals are to optimally distribute the projected traffic over the network resources by minimizing the overall cost and at the same time avoid overloading parts of the network while others are under-loaded. In general, this problem can be formulated as a network flow optimisation problem [12]. We defined the cost of each link as the sum of linear functions $f_h(x_{l,h})$ per PHB, where $x_{l,h}$ is the load on the link l from PHB h. The total cost should be the sum of f_h for all PHBs over all links; this is the objective function to be minimized.

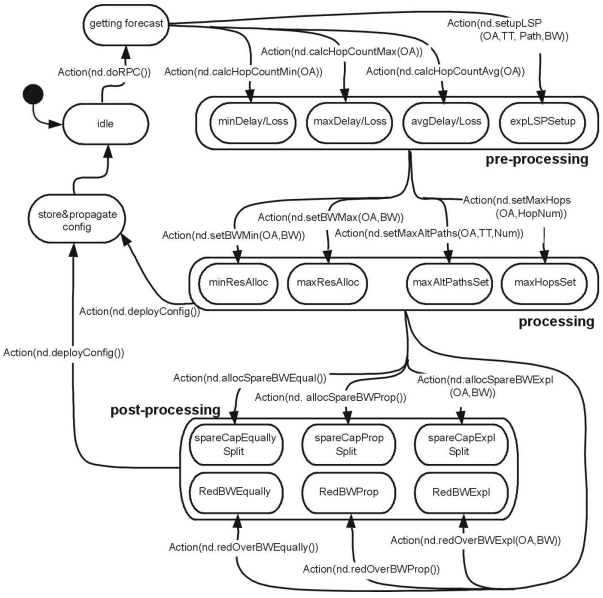

Figure 1: Network Dimensioning module behaviour.

As it can be seen from the behavioural state chart in Figure 1, ND goes through 3 main states in order to produce a network configuration, namely the *pre-processing*, *processing* and *post-processing* state. The details of the actions supported by the Managed Objects (MOs) of ND in every state shown in the figure are explained in detail in the following section.

3. Policies for ND

The Network Dimensioning module, besides providing long-term guidelines for sharing the network resources, can also be policy based so that its behaviour can be modified dynamically at runtime, reflecting high-level business objectives.

ND is triggered by time rather than network state events from within the network. Two categories of policies are identified for this static off-line resource management component:

i) Policies that result in providing initial values to variables, which are essential for the functionality of ND and do not depend on any state but just reflect decisions of the policy administrator (initialisation policies).

ii) Policies that depend on the input from the traffic forecast module concerning the predicted volume of traffic the produced configuration should satisfy (resource provisioning policies). Such policies are those whose execution is based on the type of traffic and on the resulting configuration of the network.

This section describes the methods supported by the key components of the module, as well as the representation of these methods as policies, which can influence its functionality.

3.1. Explicit route setup and BW allocation

This component offers methods that can explicitly define Label Switched Paths that traffic trunks should follow and also explicitly define the way the BW should be allocated to different OAs. The following methods are supported by this component's MOs:

```
setBWMin(OA, BW)                                    (M1.1)
setBWMax(OA, BW)                                    (M1.2)
setupLSP(OA, [TT], [PATH], BW)                        (M2)
```

The first two methods (`M1.1` and `M1.2`) allow the administrator to define the amount of network resources (giving a minimum, maximum or a range) to be allocated to each OA. The BW value is expressed as a percentage of the overall network capacity. Method `M2` provides the ability to explicitly define an LSP for traffic that belongs to a particular OA and passes through the set of nodes defined by `PATH` with logically assigned bandwidth, `BW`.

3.2. Hop count derivation

Another important function of ND is to handle the QoS requirements of the expected traffic in terms of delay and packet loss. In our implementation of ND functionality, we simplify our optimisation problem by transforming the delay and loss requirements into constraints for the maximum hop count for each traffic

trunk. This transformation is made possible by keeping statistics for the delay and loss rate of the PHBs per link. The accuracy of the statistics is determined by the period used to obtain them and smoothing methods such as exponential weighted moving average can be used. The following methods are offered by this component's MOs:

```
calcHopCountMin(OA)                    (M3.1)
calcHopCountMax(OA)                    (M3.2)
calcHopCountAvg(OA)                    (M3.3)
```

These methods define the way to derive the hop count constraint for every OA. Different options can be selected by using the minimum, maximum, or average in order to derive the maximum hop count constraint. We envisage that by using the maximum we are too conservative (appropriate for EF traffic), while by using an average we possibly underestimate the QoS requirements.

3.3. Optimisation algorithm

This is the core component of Network Dimensioning and contains the optimisation algorithm. Its objective is to find a set of paths for which the bandwidth requirements of the traffic trunks are met, the delay and loss requirements are met by using the hop count constraint as an upper bound and the overall cost function is minimised [8]. The methods defined below are offered by this component's MOs for setting parameters that influence the way the algorithm calculates the output configuration:

```
setMaxAltPaths(OA, [TT], PathNum)      (M4)
setMaxHops(OA, HopNum)                 (M5)
```

The first method sets an upper bound on the number of hops the calculated paths are permitted to have. This number may vary depending on the class (OA) the traffic belongs to. The second method defines the number of alternative paths the optimisation algorithm should define for every traffic trunk that belongs to the defined OA or even for a specific trunk, for the purpose of load balancing.

3.4. Spare/Over-provisioned BW treatment

After the dimensioning algorithm finishes, ND enters a post-processing stage where it will try to assign the residual physical capacity to the various traffic classes or reduce the allocated capacity because the link capacity is not enough to satisfy the predicted traffic requirements. The following methods are offered by this component's MOs:

```
allocSpareBWEqual( )                   (M6.1)
allocSpareBWProp( )                    (M6.2)
allocSpareBWExlp(OA, BW)               (M6.3)
redOverBWEqual( )                      (M7.1)
redOverBWProp( )                       (M7.2)
redOverBWExpl(OA, BW)                  (M7.3)
```

Method `M6` defines the distribution of spare capacity for every OA. The distribution can be done equally between the OAs, proportionally to the current allocation or explicitly, where the amount of BW is specified as a percentage. Method `M7` is similar to the previous one, defining the amount of bandwidth to be reduced with regard to an OA, in order to fit the physical link capacity.

3.5. Policy representation

Extended research on policy-based systems identified several types of policies that are useful for managing distributed systems [5]. Obligation policies fall in the category of management policies and are of particular interest to our work. They 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.

The methods supported by the different ND components described in the previous sections can be used to encode the action part of an obligation policy that follows the format provided by the Ponder specification language [5]. In the context of this work the subject for all ND related policies is a management entity known as the ND PMA (Policy Management Agent). The example that follows encodes method `M1.1` in the policy specification:

```
inst oblig /policies/nd/PolA {
          on    doNDProc;
          subj  s = ndPMA;
          targ  t = nd/baMO/network;
          do    t.setBWMin(OA, BW);
          when  constraints;
}
```

We define three types of events, namely the *pre-processing*, *processing* and *post-processing* event. Each of the events is responsible for triggering the appropriate policies related to the different stages of the dimensioning process, as in the example, where *PolA* is triggered by the system event *processing*. The policy targets are the specific MOs provided by the different ND components supporting the relevant methods. In the example representation, the target is defined as a sub-domain of the BW allocation MO, which provides a logical representation of the underlying network. Additional constraints can be specified to define any further conditions that have to be met, like the time period for which the policy is valid. This constraint can be useful when the administrator needs to specify a different network configuration for busy or non-busy hours of the day.

4. ND policy conflicts classification

The fact that policies are downloaded to the ND module 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. We have identified a number of potential conflicts related to obligation policies that guide the ND functionality, and classified them as shown in Figure 2.

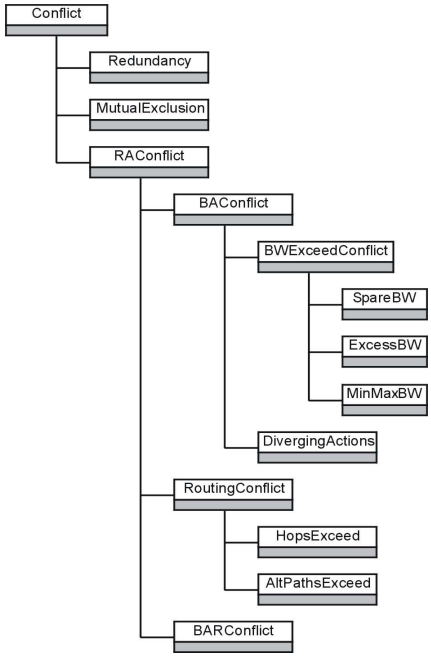

Figure 2: ND conflict classification.

The first two categories – *redundancy* and *mutual exclusion* – involve conflicts that are domain-independent and apply to any policy driven system. The rest are application-specific conflicts, related to QoS resource management policies that are responsible for BW allocation to different OAs or QoS classes, and route (LSP) calculation as well as setup. We term these conflicts as resource allocation conflicts (*RAConflicts*), which can be subdivided to BW allocation conflicts (*BAConflicts*), routing conflicts (*RoutingConflicts*) and a combination of the last two, BW allocation and routing conflicts (*BARConflicts*). This section describes the conditions under which the specified types of conflicts would arise.

4.1. Redundancy conflicts

Redundancy conflicts may arise because of duplicate policies or policies with inconsistent action parameters in relation to others. If two policies are characterized by the same subjects, targets, actions and action parameters, they are said to be duplicate and should not be allowed to coexist. 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 policy instances of *PolA* where the first action specifies that at least 30% of the resources should be allocated for EF traffic and the second 40% for the same traffic type.

```
do    t.setBWMin(ef, 30%)
do    t.setBWMin(ef, 40%)
```

In this case the OA parameter is the key parameter matched, signifying that the two actions will lead to a redundancy conflict irrespective of the associated BW value.

4.2. Mutual Exclusion (ME) conflicts

The functionality of the ND components allows for a choice of methods related to a specific process, i.e. different strategies for realising a goal. Such process is, for example, the allocation of spare resources, where the remaining network capacity after the *processing* stage is assigned equally, proportionally or explicitly between the various OAs. The different actions are said to be mutually exclusive since there should not be more than one directive specifying how spare resources are to be allocated. Therefore, two policies will result in a conflict if their actions are mutually exclusive. Table 2 summarises the identified actions that are mutually exclusive.

Table 2: Sets of mutually exclusive actions.

Spare BW allocation	Over-provisioned BW reduction	Hop count derivation
`allocSpareBWEqual()`	`redOverBWEqual()`	`calcHopCountMin(OA)`
`allocSpareBWProp()`	`redOverBWProp()`	`calcHopCountMax(OA)`
`allocSpareBWExlp(OA,BW)`	`redOverBWExpl(OA,BW)`	`calcHopCountAvg(OA)`

4.3. BW allocation conflicts

BW allocation conflicts relate to the way the ND module assigns link capacity to the different OAs. This conflict type is subdivided into *DivergingActions* and *BWExceed* conflicts, which arise due to the existence of specific actions with inconsistent parameter values with respect to one another.

With a combination of `setBWMin` and `setBWMax` actions, the administrator can specify a range of network resources to be allocated to the various OAs (Figure 3a). The following two actions aim to provide such a range for EF traffic between `BW1` and `BW2`:

```
do    t.setBWMin(ef, BW1)
do    t.setBWMax(ef, BW2)
```

If the above actions are encoded into two separate policies with the same subjects and targets, there is a possibility that the BW values specified in the actions will not converge to provide the intended BW range. Instead, the values are said to be diverging if `BW1>BW2`,

in which case a *DivergingActions* conflict should be signalled (Figure 3b).

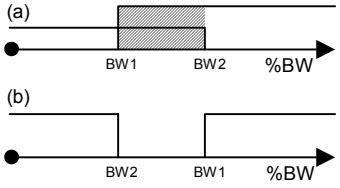

Figure 3: BW allocation (min-max).

In addition to specifying how the network resources are assigned as a whole, the above actions can be used to allocate the BW of specific links (members of the network domain). The administrator may decide that for a critical link, the allocation between the various OAs passing through that link should be explicitly specified, where a critical link can be defined as one of high importance either because of its location or its heavy loading with certain types of traffic. The same principle for a conflict applies when the target of both actions is a specific link or when one of the actions targets a specific link and the other the network as a whole. As a general rule, a *DivergingActions* conflict will arise between two policies if they have the same or overlapping targets and diverging actions with matching OA parameters but inconsistent BW values.

During the post-processing stage the administrator can define how any spare BW will be shared among the OAs, or how over-provisioned BW is to be reduced. If this process is carried out using explicit actions, there is a potential that the sum of the specified BWs for the various OAs may exceed the allowed value of 100% due to human error.

```
do    t.allocSpareBWExlp(ef, BW1)
do    t.allocSpareBWExlp(af, BW2)
do    t.allocSpareBWExlp(be, BW3)
```

For the above example actions a *BWExceed* conflict will occur if `BW1+BW2+BW3>100%`. The same rule applies to explicit actions responsible for the reduction of excess BW, as well as to `setBWMin` and `setBWMax` actions in the ND *processing* stage. One could argue that this inconsistency falls in the area of validation, where the underlying system cannot support the policy requirements. We consider it a conflict because it depends on the action parameters of more than one policy.

4.4. Routing conflicts

Routing conflicts relate to the way the ND module assigns routes that TTs should follow and the specification of the maximum number of hops or alternative paths an OA should have in order to meet the QoS characteristics. This conflict type is subdivided into *HopsExceed* and *AltPathsExceed* conflicts, which arise due to the existence of specific actions with inconsistent parameter values. There is a potential *HopsExceed* conflict between two policies with the following actions:

```
do    t.setMaxHops(OA, HopNum)
do    t.setupLSP(OA, [TT], [PATH], BW)
```

The conflict will occur if the hop-count of the `PATH` parameter in the `setupLSP` action exceeds the maximum number of allowed hops specified in the first action, for the same OA.

The second conflict related to routing policies will arise between the two actions below, if the number of instantiated policy actions of type `setupLSP` exceeds the maximum number of allowed alternative paths specified in the first action, for the same OA and TT.

```
do    t.setMaxAltPaths(OA, [TT], PathNum)
do    t.setupLSP(OA, [TT], [PATH], BW)
```

4.5. BAR conflicts

The resource allocation conflicts identified are specific to either BW allocation or routing. *BAR* conflicts are related to both the previous two and will occur between two policies with actions as stated below, if the BW parameter of the `setupLSP` action is greater than the maximum allowed BW by the first policy action, i.e. `BW2>BW1`, for the same OA.

```
do    t.setBWMax(OA, BW1)
do    t.setupLSP(OA, [TT], [PATH], BW2)
```

Furthermore, the conditions for a *BAR* conflict will be satisfied if there is an overlap between any of the nodes defined in the `PATH` parameter and the target of the first action.

5. Conflict analysis

According to the description of the conditions under which a conflict in the policy specifications may arise, specific rules can be defined to detect such an event. For the process of conflict detection we follow the approach presented in [3], where both the rules and the policies are expressed in EC notation. For example, the representation of the obligation policy *PolA* in EC would be as follows:

```
initiates(sysEvent(doNDProc), oblig(polA, ndPMA,
          operation(network, setBWMin(OA, BW))), T) ←
          constraints.
```

The rules are expressed in the form of logic predicates that encapsulate the conditions that have to be met to signal a conflict. These predicates are used as conflict fluents in EC notation and can be considered as goal states that, when they are achieved, 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.

5.1. System architecture

The system architecture is presented in Figure 4 and can be considered as a decomposition of a policy management tool. Our approach towards conflict detection is based on the output of the refinement process, where high-level policy specifications introduced in the policy creation environment are decomposed into low-level implementable ones and mapped to their respective EC representation. This process makes use of the domain hierarchy for the managed objects, as well as information about the system behaviour in the form of state charts.

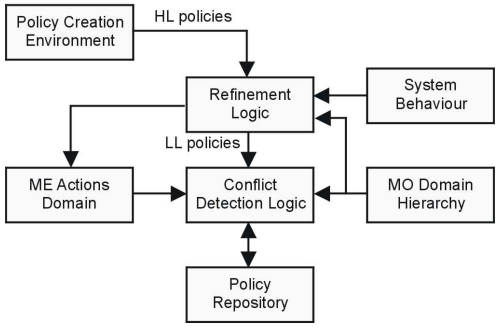

Figure 4: System architecture.

According to our work on policy refinement [13], in some cases the process of goal elaboration yields a disjunction of goals from which the user can select the sub-goal that best satisfies the requirement. The actions that achieve the different goals are said to be mutually exclusive (ME), and can be classified into appropriate domains. This information together with the managed objects domain hierarchy are fed as input to the detection process, where the necessary logic is applied to a pool of low-level policies to determine if there are any domain-independent or application-specific conflicts between them. Conflict-free policies are stored in the repository. Note that the communication between detection logic and the repository is bi-directional signifying that we not only aim to detect conflicts that may exist between new policies from the output of the refinement process, but also between new policies and ones already stored in the repository.

5.2. Domain-independent conflicts

The detection process regarding domain-independent conflicts requires mainly information provided by the policy specification. This information can be used to express the conditions under which specific predicates should signal a conflict.

In order to capture *redundancy* conflicts as defined in section 4.1, we have to cater for all types of actions and their associated parameters (represented as lists). Based on the number of action parameters, the `redundancyConflict` predicate performs checks that involve list manipulation, and aims to match certain key parameters as well as actions to signal the occurrence of a conflict:

```
holdsAt(redundancyConflict(PolID1, PolID2,
                           Action1, Action2), T) ←
    holdsAt(oblig(PolID1, Subj,
            operation(Targ, Action1(Params1))), T) ∧
    holdsAt(oblig(PolID2, Sub,
            operation(Targ, Action2(Params2))), T) ∧
    (emptyLists(Params1, Params2) ∧ Action1 == Action2) ∨
    (listMemberCount1(Params1, Params2) ∧
     firstMember(Params1) ==  firstMember(Params2) ∧
     Action1 == Action2) ∨
    (listMemberCount2(Params1, Params2) ∧
     firstMember(Params1) ==  firstMember(Params2) ∧
     Action1 == Action2) ∨
    (listMemberCount3(Params1, Params2) ∧
     firstMember(Params1) ==  firstMember(Params2) ∧
     secondMember(Params1) == secondMember(Params2) ∧
     Action1 == Action2) ∨
    (listMemberCount4(Params1, Params2) ∧
     firstMember(Params1) ==  firstMember(Params2) ∧
     secondMember(Params1) == secondMember(Params2) ∧
     thirdMember(Params1) == thirdMember(Params2) ∧
     Action1 == Action2).
```

The ME actions defined in section 4.2 are identified by the refinement process in [13] and classified into domains (Figure 5).

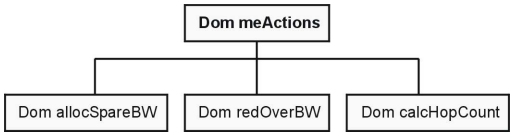

Figure 5: ME actions domain membership.

The detection process for an ME conflict between two actions involves identifying their membership in the ME actions domain – if they belong to the same domain there is a potential conflict. Any actions with domain *allocSpareBW* or *redOverBW* membership are conflicting by default, but for domain *calcHopCount* membership a conflict will arise if there is also a match between the OA action parameters of the two policies. The `meConflict` predicate is responsible for detecting an ME conflict as defined below:

```
holdsAt(meConflict(PolID1, PolID2, Action1, Action2),T) ←
    holdsAt(oblig(PolID1, Subj,
            operation(Targ, Action1(Params1))), T) ∧
    holdsAt(oblig(PolID2, Subj,
            operation(Targ, Action2(Params2))), T) ∧
    (isMember(Action1, domAllocSpareBW) ∧
     isMember(Action2, domAllocSpareBW)) ∨
    (isMember(Action1, domRedOverBW) ∧
     isMember(Action2, domRedOverBW)) ∨
    (isMember(Action1, domCalcHopCount) ∧
     isMember(Action2, domCalcHopCount) ∧
     Params1 == Params2).
```

5.3. Application-specific conflicts

The detection process for application-specific conflicts requires not only information provided by the policy specification, but also application-specific information. In the context of our work, 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 or more policies violate these constraints.

The `dvrgActionsConflict` predicate as defined below, indicates a conflict between policies related to BW allocation during the *processing* stage of ND. Here, the constraints conveyed to the conditional part of the predicate include the specific policy actions with matching OA parameters and inconsistent BW values, as well as matching or overlapping targets. The final domain membership relation caters for the condition where one policy targets the network as a whole, and the other a specific link.

```
holdsAt(dvrgActionsConflict(PolID1, PolID2, BW1, BW2), T) ←
    holdsAt(oblig(PolID1, Subj,
            operation(Targ1, setBWMin(OA1, BW1))), T) ∧
    holdsAt(oblig(PolID2, Subj,
            operation(Targ2, setBWMax(OA2, BW2))), T) ∧
    (OA1 == OA2) ∧ (BW1 > BW2) ∧ (Targ1 == Targ2 ∨
    isMember(Targ1, Targ2) ∨ isMember(Targ2, Targ1)).
```

The `bwExcdSpareConflict` predicate below assumes that there are three types of OAs available: Expedited Forwarding, Assured Forwarding and Best Effort. This rule will signal a conflict related to policies that explicitly define how spare BW is split among the three OA types, during the post-processing stage of ND. The inconsistency detected here is when the sum of the specified BW parameters exceeds the value of 100%. The same principle applies to policies responsible for explicit reduction of excess BW and to policies for explicit BW allocation.

```
holdsAt(bwExcdSpareConflict(PolID1, PolID2, PolID3,
                            BW1, BW2, BW3), T) ←
    holdsAt(oblig(PolID1, Subj,
          operation(Targ, allocSpareBWExlp(ef, BW1))), T) ∧
    holdsAt(oblig(PolID2, Subj,
          operation(Targ, allocSpareBWExlp(af, BW2))), T) ∧
    holdsAt(oblig(PolID3, Subj,
          operation(Targ, allocSpareBWExlp(be, BW3))), T) ∧
    sumOf(BW1, BW2, BW3) > 100.
```

The rest of the identified conflicts are detected in a similar manner, encoding the application specific constraints in the conditional fields of specific predicates. The relevant predicates are presented below and they follow the guidelines provided in section 4.

```
holdsAt(hopsExcdConflict(PolID1, PolID2, PATH, HopNum), T) ←
    holdsAt(oblig(PolID1, Subj,
          operation(Targ, setMaxHops(OA1, HopNum))), T) ∧
    holdsAt(oblig(PolID2, Subj,
          operation(Targ, setupLSP(OA2, TT, PATH, BW))), T) ∧
    (OA1 == OA2) ∧ (hopCount(PATH) > HopNum).

holdsAt(pathsExcdConflict(PolID1, PolID2), T) ←
    holdsAt(oblig(PolID1, Subj, operation(Targ,
            setMaxAltPaths(OA1, TT1, PathNum))), T) ∧
    holdsAt(oblig(PolID2, Subj, operation(Targ,
            setupLSP(OA2, TT2, PATH, BW))), T) ∧
    (OA1 == OA2) ∧ (TT1 == TT2) ∧
    instCount(operation(Targ,
            setupLSP(OA2, TT2, PATH, BW))) > PathNum.

holdsAt(barConflict(PolID1, PolID2, BW1, BW2), T) ←
    holdsAt(oblig(PolID1, Subj,
            operation(Targ1, setBWMax(OA1, BW1))), T) ∧
    holdsAt(oblig(PolID2, Subj, operation(Targ2,
            setupLSP(OA2, TT, PATH, BW2))), T) ∧
    (OA1 == OA2) ∧ (BW2 > BW1) ∧
      (isMember(Targ1, PATH) ∨
       isMember(PATH, Targ1)).
```

5.4. Conflict detection examples

By using one of the conflict fluents (e.g. `meConflict`) as a goal state of an abductive query, it is possible 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 this particular conflict type.

We have developed a tool that uses the A-System abductive proof engine together with SICStus Prolog [14] for detecting the identified conflict types. The tool takes as input the policy specifications, applies the appropriate detection logic and provides the user with a command line interface to query the system for any domain-independent or application-specific conflicts that may exist. Consider the following pool of policies in their EC representation:

```
initiates(sysEvent(doNDPreProc), oblig(p1, ndPMA,
          operation(hopCountMO, calcHopCountMin(af))), T) :-
          between(9,0,0,10,0,0, T), time(T).

initiates(sysEvent(doNDPreProc), oblig(p2, ndPMA,
          operation(hopCountMO, calcHopCountAvg(af))), T) :-
          between(9,30,0,10,30,0, T), time(T).

initiates(sysEvent(doNDProc), oblig(p3, ndPMA,
          operation(network, setBWMin(ef, 50))), T) :-
          between(16,0,0,20,0,0, T), time(T).

initiates(sysEvent(doNDProc), oblig(p4, ndPMA,
          operation(network, setBWMax(ef, 40))), T) :-
          between(18,0,0,22,0,0, T), time(T).

initiates(sysEvent(doNDPreProc), oblig(p5, ndPMA,
          operation(optMO, setMaxHops(ef, 4))), T) :-
          between(13,0,0,19,0,0, T), time(T).

initiates(sysEvent(doNDPreProc), oblig(p6, ndPMA,
          operation(lspMO, setupLSP(ef, [r2,r15],
                    [r2,r4,r6,r8,r9,r11,r15], 45))), T) :-
          between(9,30,0,18,30,0, T), time(T).

initiates(sysEvent(doNDProc), oblig(p7, ndPMA,
          operation(network, setBWMin, parms(af, 60))), T) :-
          between(16,0,0,20,0,0, T), time(T).

initiates(sysEvent(doNDProc), oblig(p8, ndPMA,
          operation(network, setBWMax, parms(af, 50))), T) :-
          between(20,00,0,22,0,0, T), time(T).
```

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(af)` 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.

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. To demonstrate the above we provide the output of several queries to the tool:

```
?- solve(conflict(Type, ConflictData, T)).
  Solution found
  abduced atoms:
          0-happens(clocktick(9,0,0), 0)
          1-happens(clocktick(9,30,0), 1)
          2-happens(sysEvent(doNDPreProc), 2)
```

```
        3-happens(clocktick(10,0,0), 3)
        4-happens(clocktick(10,30,0), 4)
Solved query:
    conflict(meConflict, conflictData(p2, p1,
            calcHopCountAvg, calcHopCountMin), 3)

Solution found
abduced atoms:
        0-happens(clocktick(16,0,0), 0)
        1-happens(clocktick(18,0,0), 1)
        2-happens(sysEvent(doNDProc), 2)
        3-happens(clocktick(20,0,0), 3)
        4-happens(clocktick(22,0,0), 4)
Solved query:
    conflict(dvrgActionsConflict,
            conflictData(p3, p4, 50, 40), 3)

Solution found
abduced atoms:
        0-happens(clocktick(9,30,0), 0)
        1-happens(clocktick(13,0,0), 1)
        2-happens(sysEvent(doNDPreProc), 2)
        3-happens(clocktick(13,30,0), 3)
        4-happens(clocktick(18,0,0), 4)
Solved query:
    conflict(hopsExcdConflict, conflictData(p5, p6,
            [r2,r4,r6,r8,r9,r11,r15], 4), 3)

Solution found
abduced atoms:
        0-happens(clocktick(9,30,0), 0)
        1-happens(clocktick(18,0,0), 1)
        2-happens(sysEvent(doNDProc), 2)
        3-happens(sysEvent(doNDPreProc), 2)
        4-happens(clocktick(18,30,0), 3)
        5-happens(clocktick(22,0,0), 4)
Solved query:
    conflict(barConflict, conflictData(p4, p6, 40, 45), 3)
```

The results suggest that there is an ME conflict between `P1` and `P2` because of mutually exclusive actions, a BA conflict between `P3` and `P4` because of inconsistent BW values, a routing conflict between `P5` and `P6` because the hop-count of the specified path exceeds the maximum number of hops allowed, and a BAR conflict between `P4` and `P6` because the BW allocated is more than the maximum allowed for EF traffic. Additionally the results describe the sequence of events that need to take place for the conflict to occur. Notice 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.

6. Related work

Research in conflict analysis has been actively growing over the years, but most of the work in this area addresses general management policies. The authors in [15] classify conflicts as domain-independent and application-specific, and in [10] the authors identify application-specific conflicts like conflicts of duty, conflicts of priorities for resources and self-management conflicts.

Among the many alternative approaches to policy specification, there are a number of proposals for formal, logic-based notations. In particular, logic-based languages have proved attractive for the specification of security policy, as they support a well-understood formalism, amenable to analysis. However, they can be difficult to use and are not always directly translatable into efficient implementation. One such example is the Policy Description Language (PDL) [16], which is used for the specification of obligation policies. The language can be described as a real-time specialized production rule system to define policies. The syntax of PDL is simple and policies are described by a collection of two types of expressions: *policy rules* and *policy defined event propositions*. Later work by Chomicki [17], extends PDL to include the concept of action constrains, which are policies that prevent a specified action from being performed in a given situation. This work introduces the idea of using a policy monitor to detect conflict situations and resolve them by either suppressing the events that could lead to a conflict or overriding the conflicting action. Additionally, work by Son and Lobo, presents an approach for reasoning about policies with the objective of mapping a desired action history back to a possible event history [18]. This work is interesting because it illustrates how formal techniques together with logic programming can be used to derive information about the policy program – in this case the event history that causes a particular set of actions.

One of the few conflict analysis examples that targets a specific application domain is presented in [19], where all possible firewall rule relations have been formally defined and were used to classify firewall policy anomalies. The tool developed in the context of this work, called the *Firewall Policy Advisor*, can detect the presence of anomalies in the policy specification and prompt the administrator to make the necessary changes.

7. Conclusions and future work

In this paper we indicated the types of application-specific potential conflicts that may arise during policy specification using off-line Network Dimensioning for QoS management as a case study. We classified these conflicts into domain-independent and application-specific, and specified the conditions under which these conflicts may arise. The formal language of Event Calculus was used to analyse the policy specification by defining the rules for conflict detection, and abduction provided the means to not only identify a conflict but to also provide an explanation as to how that conflict occurred. Finally, we showed conflict detection examples from our initial implementation of a conflict analysis tool. The case study provides an example of the application-specific analysis needed to determine potential conflicts and how to formalise them to automate the conflict detection.

We term the identified conflicts as intra-component conflicts since they are specific to policies applied to a single module of the TEQUILA architecture. Part of our future work will involve the classification and

detection of possible intra-component conflicts related to the rest of the TEQUILA modules, such as the SLS Subscription, and Dynamic Resource Management. Also, due to the hierarchical relationship between policies defined for the different modules, there is a need to detect possible inconsistencies that may arise between policies specified for different layers. We term this inter-component conflict detection.

It is highly possible that certain conflicts may depend on the runtime state of the system. Thus, besides the detection of static intra and inter-component static conflicts we plan to extend our work to dynamic or run-time conflict detection. Finally, we aim to provide a mechanism for automated conflict resolution through the use of meta-policies, where specific rules will be defined to specify which of the conflicting policies will prevail.

Acknowledgements

The work presented in this paper was carried out in the context of the EPSRC PAQMAN project (Policy Analysis for Quality of service MANagement) - grant numbers GR/R31409/01 and GR/S79985/01.

References

[1] E.C. Lupu and M.S. Sloman, "Conflicts in Policy-Based Distributed Systems Management, " *In IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management*, vol. 25, pp. 852-869, 1999.

[2] S. Jajodia, P. Samarati, and V.S. Subrahmanian, "A Logical Language for Expressing Authorisations, " presented at IEEE Symposium on Security and Privacy, Oakland, USA, 1997a.

[3] A.K. Bandara, E.C. Lupu, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," presented at 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2003), Lake Como, Italy, 2003.

[4] R.A. Kowalski and M.J. Sergot, "A logc-based calculus of events," *New Generation Computing*, vol. 4, pp. 67-95, 1986.

[5] N. Damianou, N. Dulay, E.C. Lupu, and M.S. Sloman, "The Ponder Policy Specification Language," presented at 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2001), Bristol, UK, 2001.

[6] A.C. Kakas, R.A. Kowalski, and F. Toni, "The Role of Abduction in Logic Programming," Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 235-324, 1998.

[7] P. Flegkas, P. Trimintzios, and G. Pavlou, "A Policy-based Quality of Service Management Architecture for IP DiffServ Networks," IEEE Network Magazine Special Issue on Policy Based Networking, vol. 16 No. 2, pp. 50-56, 2002.

[8] P. Flegkas, P. Trimintzios, G. Pavlou, and A. Liotta, "Design and Implementation of a Policy-Based Resource Management Architecture" presented at IEEE/IFIP Integrated Management Symposium (IM 2003), Colorado Springs, Colorado, USA, Kluwer, pp. 215-229, 2003.

[9] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer, "An Abductive Approach for Analysing Event-Based Requirements Specifications, " presented at 18th Int. Conf. on Logic Programming (ICLP), Copenhagen, Denmark, 2002.

[10] J.D. Moffett and M.S. Sloman, "Policy Conflict Analysis in Distributed System Management, " *Journal of Organisational Computing*, vol. 4, pp. 1-22, 1994.

[11] K. Nichols, V. Jacobson and L. Zhang, "A Two Bit Differentiated Services Architecture for the Internet," in *Network Working Group – RFC2638,* http://www.ietf.org/rfc/rfc2638.txt, 1999.

[12] R. Ahuja, T. Magnanti, and J. Orlin, "Network Flows: Theory, Algorithms and Applications," Prentice-Hall, 1993.

[13] A.K. Bandara et al., "Policy Refinement for DiffServ Quality of Service Management," accepted for the proceedings of IEEE/IFIP Integrated Management Symposium (IM 2005), Nice, France, 2005.

[14] B. van Nuffelen and A. Kakas, "A-System: Programming with abduction," presented at Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), 2001.

[15] E.C. Lupu and M.S. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *In IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management*, vol. 25, pp. 852-869, 1999.

[16] J. Lobo, R. Bhatia, and S. Naqvi, "A Policy Description Language," presented at 16th National Conf. on Artificial Intelligence, Orlando, Florida, USA, 1999.

[17] J. Chomicki, J. Lobo, and S. Naqvi, "A Logic Programming Approach to Conflict Resolution in Policy Management," presented at 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA, 2000.

[18] T.C. Son and J. Lobo, "Reasoning about Policies Using Logic Programs," presented at AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, Stanford University, CA, 2001.

[19] E. Al-Shaer and H. Hamed, "Modeling and Management of Firewall Policies," in IEEE Transactions on Network and Service Management (eTNSM 2004), Volume 1-1, April 2004.