Dynamic Policy Analysis and Conflict Resolution for DiffServ Quality of Service Management

Marinos Charalambides, Paris Flegkas,
George Pavlou, Javier Rubio-Loyola
Centre for Communications Systems Research
University of Surrey
Guilford, GU2 7XH, UK
{m.charalambides, p.flegkas, g.pavlou,
j.rubio-loyola}@eim.surrey.ac.uk

Arosha K Bandara, Emil C Lupu, Alessandra Russo,
Morris Sloman, Naranker Dulay
Department of Computing, Imperial College London
South Kensington Campus
London, SW7 2AZ, UK
{a.k.bandara, e.c.lupu, a.russo, m.sloman,
n.dulay}@imperial.ac.uk

Abstract— **Policy-based dynamic resource management may involve interaction between independent decision-making components which can lead to conflicts. For example, conflicts can occur between the policies for allocating resources and those setting quotas for users or classes of service. These policy conflicts cannot be detected by static analysis of the policies at specification-time as the conflicts arise from the current state of the resources within the system and so can only be detected at run-time. In this paper we use policies related to Quality of Service (QoS) provisioning for configuring Differentiated Services (DiffServ) networks to illustrate techniques for the dynamic detection and resolution of conflicts. Configuration includes implementing network provisioning decisions, performing admission control, and adapting bandwidth allocation dynamically according to emerging traffic demands. We identify possible conflicts between policies that manage the allocation of resources, and we also investigate conflicts that may arise between these policies and higher-level directives refined at the dynamic resource management level, acting as constraints. The paper shows how Event Calculus can be used to detect conflicts, focusing on the ones that emerge at run-time, and provides an approach for specifying policies to automate conflict resolution. The latter is demonstrated through our initial implementation of a dynamic conflict analysis tool.**

Keywords: Conflict detection; Conflict resolution; Policy-based resource management

I. 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, storage allocation and processing resources for server clusters. Yet despite various 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 configuration stability given that policies may have conflicts leading to unpredictable effects.

Work on policy analysis has mainly focused on conflicts that can be determined statically at compile-time [1]. The detection process involved simple policy analysis and resolution based on the specification of policy precedence rules [1,2] that may not suit many policy-driven systems. Although we believe that static analysis is very useful for detecting and resolving some conflicts before policies are deployed, it cannot detect many conflicts in resource management policies which occur as a result of the current state of the resources. For example, policies which increment or decrement allocation of resources may conflict with policies related to setting upper and lower bounds for the resources. These conflicts result from current state of the resource allocation and bounds so can only be detected and resolved at run-time.

This paper extends our previous work on static conflict analysis [3] by addressing the area of dynamic conflict detection and resolution in 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 [4]. TEQUILA uses DiffServ together with Multi-Protocol Labelled Switching (MPLS) to support a network that can dynamically adapt to varying traffic demands. More specifically, we focus on conflicts that may arise from policies driving the Dynamic Resource Management (DRsM) module of the TEQUILA framework. These are policies specified explicitly for this particular module, or higher-level directives that are refined at the DRsM level acting as constraints to its functionality.

The work in this paper is based on the work presented in [5] where the use of Event Calculus (EC) was proposed as a specialised first-order logic for formalising policy specification and the mapping to and from the Ponder policy language [6]. EC allows specification of the system behaviour using familiar notations, such as state charts, which can then be automatically

This work was carried out in the context of the EPSRC PAQMAN (grant numbers GR/R31409/01 and GR/S79985/01), and FP6 Information Society Technologies EMANICS Network of Excellence (IST-026854) research projects.

translated into the logic program representation. In this work, we identify the possible conflicts that might emerge between policies driving the behaviour of the DRsM module and classify them into domain-independent and application-specific. Using EC we specify a set of rules that define the conditions for a conflict, focusing on the ones that can only be determined at run-time. Based on the identified conflict types, we provide possible resolution strategies in the form of policies, which are enforced once a conflict is detected and can be considered as extensions to the DRsM functionality, supporting resolution logic.

In the next section, we present some background information on EC and policy analysis, as well as a description of the resource management aspects of the TEQUILA framework. Section 3 details the identified policies for dynamic resource management 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 their resolution, and in section 6 we provide an example scenario demonstrating our tool support for dynamic conflict detection and resolution. Finally, section 7 presents some related work in this field; and section 8 discusses our conclusions and future work.

II. BACKGROUND

A. Formal Representation and Event Calculus

Event Calculus 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. In the context of our work, EC serves as the basis of the formal language we have developed for describing policies and managed systems. Since its initial presentation [7], a number of variations have been presented in the literature. In this work we use the form presented in [8], 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 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.

B. 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 [9], 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(conflict(obligConflict,
        conflictData([Subj, Op])), T) ←
  holdsAt(oblig(Subj, Op), T) ∧
  holdsAt(refrain(Subj, Op),T).
```

In the case of application-specific conflicts, rules 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. The rules can include ground literals, specifying the action/target object combinations that will potentially conflict. Rules for the detection of application-specific conflicts, such as conflicts of interest, conflicts of duties and self-management conflicts can be found in [5].

C. Dynamic Resource Management

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

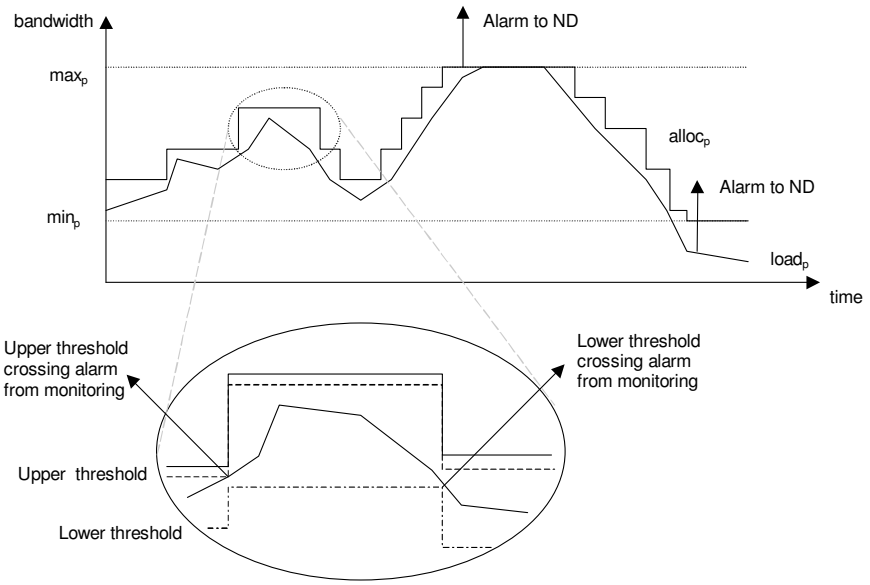

Figure 1. Bandwidth tracking of a single PHB.

detailed decomposition of the concept of a Bandwidth Broker realized as a hierarchical, logically and physically distributed system and has been presented in [4]. The Traffic Engineering (TE) sub-system of the architecture is responsible for dimensioning the underlying network according to the projected demands, and for establishing and dynamically maintaining the network configuration that has been selected to meet the QoS requirements.

The resource management aspects of TE are realised through the Network Dimensioning (ND) and Dynamic Resource Management (DRsM) modules. ND is a centralized, off-line component responsible for mapping traffic requirements to the physical network resources and for providing provisioning directives in order to accommodate the predicted traffic demand. Since the latter is based on historical data and customer subscriptions, 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 DRsM, which has distributed functionality with an instance operating in every router. It 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 (DiffServ Per-Hop Behaviours - PHBs) following the guidelines provided by ND.

Policy-based Network Dimensioning allows more flexibility in defining alternative strategies when performing an operation. For example, during the post-processing stage of ND the administrator can choose between different ways in which over-provisioned bandwidth (BW) is to be reduced to fit the physical link capacity [3]. Similarly, DRsM policies provide the flexibility to dynamically introduce logic, in the form of directives, for tracking the utilization of a PHB and ensuring that the bandwidth allocated to that PHB ($alloc_p$) is in accordance with the *required* BW. The latter is determined according to observed utilization ($load_p$). Fig. 1 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 thresholds, these values are decreased. Monitoring PHB utilizations is achieved through a monitoring component rather than polling instantaneous values. The triggering of policy actions is based on upper and lower thresholds of the BW consumed by a PHB. Monitoring will raise a threshold crossing alarm when the utilization exceeds the upper threshold or drops below the lower threshold.

III. POLICIES FOR DRSM

DRsM policies intend to manage resources allocated by ND during system operation in order to react to statistical fluctuations and special conditions that may arise. Their main objective is to guide the distribution of capacity between the PHBs defined on a link. In the rest of this section, we focus on policy actions for the calculation of new thresholds and also on actions that dictate the allocation of link BW, by managing scheduling parameters, i.e. minimum and maximum rates associated with PHBs, according to actual load conditions.

A. DRsM Components

The DRsM module comprises two components: the monitoring and the DRsM main component. The former is responsible for monitoring the PHB utilization of relevant links, issuing alarms upon upper or lower threshold crossings and calculating new thresholds. The main component is

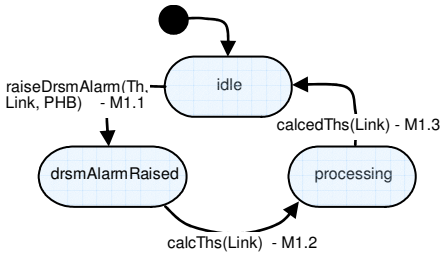

Figure 2. Monitoring component behaviour.

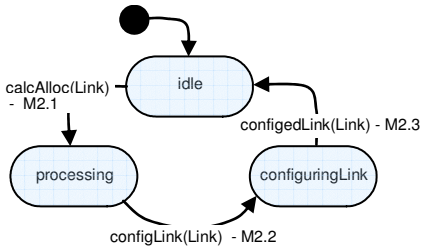

Figure 3. Main component behaviour.

responsible for the calculation of *required* BW and the allocation of that BW to the various PHBs sharing a link.

Figs. 2 and 3 depict the behaviour of the two components through state machine representations. The triggering of actions supported by the monitoring component MO (Managed Object) depends on the conditions/events that arise from the current PHB link utilization. For example, if the upper threshold of a particular PHB is exceeded, an alarm will be raised (`M1.1`) that causes new thresholds to be calculated (`M1.2`). This alarm acts as a trigger for the main component functionality (main component MO - `M2.1`), which in turn generates a new *required* BW for the PHBs sharing the link. Once the calculated BW is configured (`M2.2`), the component returns to the *idle* state (`M2.3`). The monitoring component goes *idle* after new thresholds have been calculated (`M1.3`).

B. DRsM Extended Functionality

In the previous section we provided an explanation of the basic functionality of the DRsM module. The state chart representations illustrate the simplest possible logic the module can incorporate along with actions that cause state transitions. Previous work on ND [3] demonstrated how the use of policies could extend the hard-wired functionality and also provide the policy administrator with more strategies when performing an operation. Similar logic extensions can be specified for the DRsM module as described in this section.

During the *processing* stage of both components, DRsM aims to provide new values for all the PHBs on a particular link based on a specific algorithm that takes into account PHB priorities as well as trend analysis of historical data. Alternatively, this could be achieved through explicit actions that only apply to the PHB the alarm was raised for. This means that each PHB is treated independently through appropriate methods that increase or decrease the thresholds/allocation upon upper/lower threshold-crossing alarms:

```
incrThs(Link, PHB, BW)                    (M1.21)
decrThs(Link, PHB, BW)                    (M1.22)
incrAlloc(Link, PHB, BW)                  (M2.11)
decrAlloc(Link, PHB, BW)                  (M2.12)
```

Further logic can be introduced by providing more options as to how the new values are to be calculated. For example, when an upper threshold has been crossed the administrator can opt for the allocation to be increased by an absolute value (kbps), a relative value (e.g. %5), or by using a specific algorithm. 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, the number of historical data to be used in the extrapolation function, etc. The relevant methods for the above process are as follows:

```
incrThsAbs(Link, PHB, BW)                 (M1.211)
incrThsRel(Link, PHB, BW)                 (M1.212)
incrThsAlg(Link, PHB, [params])           (M1.213)
incrAllocAbs(Link, PHB, BW)               (M2.111)
incrAllocRel(Link, PHB, BW)               (M2.112)
incrAllocAlg(Link, PHB, [params])         (M2.113)
```

C. Policy Representation

Extended research on policy-based systems identified several types of policies that are useful for managing distributed systems [6]. 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 two DRsM components described in the previous section can be used to encode the action part of an obligation policy that follows the format provided by the Ponder specification language [6]. In the context of this work, the subject for all DRsM related policies is a management entity known as the DRsM PMA (Policy Management Agent). The examples that follow encode methods `M1.211` and `M2.112` in the policy specification:

```
inst oblig /policies/drsm/PolA {
        on    drsmAlarmRaised(upprTh, link1, ef);
        subj  s = drsmPMA;
        targ  t = drsm/monitorMO;
        do    t.incrThsAbs(link1, ef, 5);
        when  constraints;
}

inst oblig /policies/drsm/PolB {
        on    drsmAlarmRaised(upprTh, link1, ef);
        subj  s = drsmPMA;
        targ  t = drsm/mainMO;
        do    t.incrAllocRel(link1, ef, 10);
        when  constraints;
}
```

The policy targets are the specific MOs, provided by DRsM, supporting the relevant methods (`monitorMO` and `mainMO`). Additional constraints can be specified to define any further conditions that have to be met, such as 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.

IV. DRsM Policy Conflicts

The fact that policies are downloaded to the DRsM 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. This section provides a taxonomy of identified conflict types and describes the conditions under which these conflicts would arise.

A. Conflict Classification

We have identified a number of potential conflicts related to obligation policies that guide the DRsM functionality, and classified them as shown in Fig. 4. Some of these conflicts can be detected using static analysis at policy specification-time, while others can only be detected at enforcement-time because they depend on the current state of the managed network and the DRsM components.

The first category, *redundancy* and *mutual exclusion*, involves 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 the allocation of BW to the different PHBs or QoS classes. These can be classified into intra and inter-module conflicts, the former being specific to policies applying to a single module of the TEQUILA architecture. Inter-module conflicts arise due to the hierarchical relationship between policies defined for different layers of the architecture, for example between ND and DRsM policies.

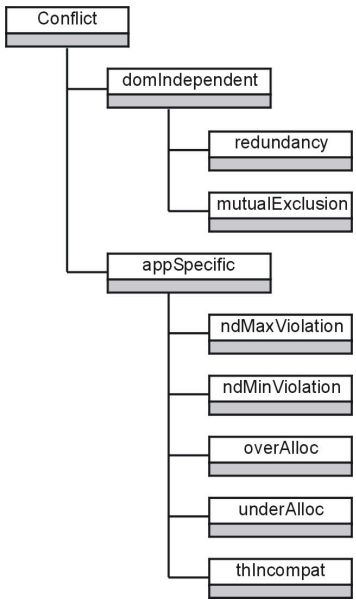

Figure 4. DRsM conflict classification.

The identified domain-independent conflicts have been presented in previous work [3] for the ND module and they also apply to DRsM. 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.

The functionality of DRsM 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 calculation of new *required* BW in the processing stage of the main component where the allocation for a specific PHB can be increased by a constant value, a relative value or using an algorithmic approach. The different actions are said to be mutually exclusive since there should not be more than one directive specifying how the allocation is to be increased. The same principle applies to policies driving the processing stage of the monitoring component. Therefore, two policies will result in a conflict if their actions are mutually exclusive.

B. Application-specific Conflicts

In a hierarchical architecture like TEQUILA, policies may be introduced at every level but higher-level policies may possibly result in the introduction of related policies at lower levels during a refinement process [10]. Thus policies for a level-*(N+1)* module may also influence the functionality of a level-*N* module.

One such relationship is between the ND and DRsM modules that constitute the main body of the TEQUILA Traffic Engineering sub-system. ND-specific policies allow the administrator to constrain the amount of network resources which can be allocated for each PHB by providing upper and lower bounds. These policies are communicated to the relevant lower-level DRsM modules during the refinement process, acting as constraints throughout the dynamic allocation of resources. This means that if a new *required* BW calculated by the DRsM main component exceeds the upper bound specified in the policy originating from ND, or drops below the lower bound, an inter-module conflict should be signalled to indicate the violation of these constraints. More specifically, an *ndMaxViolation* conflict occurs when a DRsM policy tries to increment the allocation for a specific PHB but the calculated *required* BW exceeds the upper bound, `BW1`, set by a ND-refined policy:

```
incrAlloc(Link, PHB, BW)    conflicts with
setBWMax(Link, PHB, BW1)
```

Similarly, an *ndMinViolation* conflict occurs when a DRsM policy tries to decrement the allocation for a PHB but the calculated *required* BW is less than the lower bound, `BW1`, set by a ND-refined policy:

```
decrAlloc(Link, PHB, BW)    conflicts with
setBWMin(Link, PHB, BW1)
```

Another high-level directive that is refined down to the DRsM level is a general resource management policy, which explicitly specifies that during a DRsM operational cycle, the full link capacity should be allocated between the various PHBs. This implies that a DRsM policy action aiming to increase the allocation for a specific PHB will violate the above rule since the resulting allocation will exceed the link capacity. We term this intra-module conflict as over-allocation (*overAlloc*):

```
incrAlloc(Link, PHB, BW)     conflicts with
resAlloc(Link, 100%)
```

In a similar fashion, an *underAlloc* conflict will occur when a DRsM policy aims to decrement the allocation for a particular PHB, since the resulting allocation will be less than the maximum link capacity:

```
decrAlloc(Link, PHB, BW)     conflicts with
resAlloc(Link, 100%)
```

The last of the application-specific conflicts is an intra-module conflict and involves DRsM policies responsible for the computation of new thresholds and *required* BW (as explained in section IIc). In view of the fact that these values are calculated separately by the two components, there is a potential that the allocated BW for a PHB is below its respective upper threshold, in which case a threshold incompatibility (*thIncompat*) conflict should be signalled.

V. 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 [5], where both the rules and the policies are expressed in EC notation. The occurrence of conflicts indicates the need for resolution so that DRsM can continue its operation and generate the correct values for the configuration of the underlying network. For the identified domain-independent conflicts the intervention of a human administrator is necessary, whereas for application-specific conflicts we have defined a set of policies, called *resolution policies*, which are enforced on the event of a conflict aiming to handle the situation in an automated manner.

A. Conflict Detection

Based on the identified conflict types, we have defined a set of rules that aim to signal a conflict. The rules are expressed in the form of logic predicates that encapsulate the conditions to be met for a conflict to occur. 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.

The detection process regarding domain-independent conflicts requires mainly information provided for the policy specification. This information can be used to express the conditions under which specific predicates should signal a conflict. The predicates responsible for the detection of *redundancy* and *mutual exclusion* conflicts have been presented in previous work [3]. The former aims to match certain key parameters as well as actions in the policy specification, whereas the latter makes use of mutually exclusive action domains resulting from the refinement process in [10]. Policy actions that belong to the same domain, e.g. *incrBWDom*, are conflicting and should not be allowed to coexist.

While the above conflicts can be detected through static analysis at policy specification-time, the identified application-specific conflicts can only be detected at enforcement-time depending on the state of the underlying network and the output from the processing stages of the two DRsM components. For this reason, the relevant conflict predicates require not only information provided by the policy specification, but also information regarding the run-time state of DRsM. In the context of our work, the conditions under which a conflict will arise are presented 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 `conflict(ndMaxConflict, ...)` fluent defined below indicates a violation of a ND-refined directive defining a maximum BW allocation for a PHB. Here, the constraints conveyed to the conditional part of the predicate include the specific policy actions with matching `PHB` and `Link` parameters, and the actual value of *required* BW calculated in the processing stage of the main component. The latter is represented as an argument of the *reqBW* term, the details of which are covered in section Vc. The conditions for an *ndMaxViolation* conflict will be satisfied if this value exceeds the maximum BW specified by the ND-refined policy.

```
holdsAt(conflict(ndMaxConflict, conflictData([PolID1,
                PolID2, Link, PHB, BW2, BW3])), T) ←
    holdsAt(oblig(PolID1, Subj, op(Targ,
                incrAlloc(Link, PHB, BW1))), T) ∧
    holdsAt(oblig(PolID2, Subj, op(Targ,
                setBWMax(Link, PHB, BW2))), T) ∧
    reqBW(Link, PHB, BW3) ∧
    BW3 > BW2.
```

A similar rule to the above can be specified for the *ndMinViolation* conflict, encapsulating the conditions described in section IVb. Threshold incompatibility conflicts can be detected by the `conflict(thIncompatConflict, ...)` fluent defined below. The conditions for this conflict will be satisfied if there exist policy actions for incrementing or decrementing the allocation and thresholds of a PHB, the result of which provides an inconsistent allocation with respect to the upper threshold of that PHB.

```
holdsAt(conflict(thIncompatConflict, conflictData
       ([PolID1, PolID2, PHB, BW5, ThUppr])), T) ←

    (holdsAt(oblig(PolID1, Subj, op(Targ1,
                   incrAlloc(Link, PHB, BW1))), T) ∧
     holdsAt(oblig(PolID2, Subj, op(Targ2,
                   incrThs(Link, PHB, BW2))), T)) ∨
    (holdsAt(oblig(PolID3, Subj, op(Targ1,
                   decrAlloc(Link, PHB, BW3))), T) ∧
     holdsAt(oblig(PolID4, Subj, op(Targ2,
                   decrThs(Link, PHB, BW4))), T)) ∧
    reqBW(Link, PHB, BW5) ∧
    ths(Link, PHB, ThUppr, ThLowr) ∧ (ThUppr > BW5).
```

The presence of policies regarding the full allocation of link resources during a DRsM operational cycle implies that actions for incrementing or decrementing the allocation of a PHB will induce an *overAlloc* or *underAlloc* conflict respectively. By this, we mean that after the *processing* stage of the main component, the collective allocation of the various PHBs will always violate the constraint of the high-level directive. Since these conflicts are guaranteed to occur, there is no actual need to provide relevant predicates for their detection.

B. Conflict Resolution

The process of resolving domain-independent conflicts involves giving precedence to one or more of the conflicting policies. Research on conflict resolution [1,2] identified metrics that can be used to assign priorities to conflicting policies, which can automate the conflict resolution in limited situations. However, many types of conflicts rely on human intervention for resolution. Although this process is manual, it does not impose any overheads on the functionality of the underlying DRsM modules since conflicts can be detected by static analysis before policy enforcement.

In contrast to the above, application-specific conflicts are dynamic and can only be determined at run-time, depending on the current state of the underlying network and DRsM components. This signifies the need for an automated resolution process so as to minimize the delay induced on the operation of a DRsM module when a conflict is to be resolved.

Having identified the different conflict types that may arise at run-time, the administrator can pre-specify policies that aim to provide a resolution strategy in the event of a conflict. The resolution methodology presented here does not involve identifying which of the conflicting policies will prevail, but provides separate *resolution policies* aiming to handle potential inconsistencies in our application-specific environment. These policies are triggered once the conditions for a conflict have been satisfied, extending the DRsM functionality to support resolution logic – Figs. 5 and 6.

Following the sequence with which conflicts may arise during system operation, we present the resolution policy actions we have defined for the various conflict types. After a policy for explicitly incrementing or decrementing the allocation of a particular PHB is enforced (`M1.1`/`M1.2`), a respective *overAlloc* or *underAlloc* conflict occurs by default. In this case the administrator can define how over-allocated BW will be reduced to fit the physical link capacity or how spare BW is to be shared among the various PHBs. This is achieved by methods `M1.3` and `M1.4` in Fig. 5, for which we provide different strategies:

```
redOverBWEqual(Link)                          (M1.31)
redOverBWProp(Link)                           (M1.32)
redOverBWExpl(Link, PHB, BW)                  (M1.33)
allocSpareBWEqual(Link)                       (M1.41)
allocSpareBWProp(Link)                        (M1.42)
allocSpareBWExpl(Link, PHB, BW)               (M1.43)
```

The methods above define three ways for handling these conflicts: the reduction/distribution of resources can be done equally between the PHBs, proportionally to the current allocation or explicitly, where the amount of BW is specified as a percentage. The decision on which of the strategies to use would depend on the PHB involved in the conflict and the associated link.

Once a new *required* BW is generated, there is a need to check whether the calculated value violates upper or lower constraints imposed by policies originating from ND. In the event of an *ndMinViolation* or *ndMaxViolation* conflict for a particular PHB, possible resolution actions would be to set the allocation to the value associated with the relevant bounds provided by ND:

```
setNDMax(Link, PHB, NDmax)                    (M1.5)
setNDMin(Link, PHB, NDMin)                    (M1.6)
```

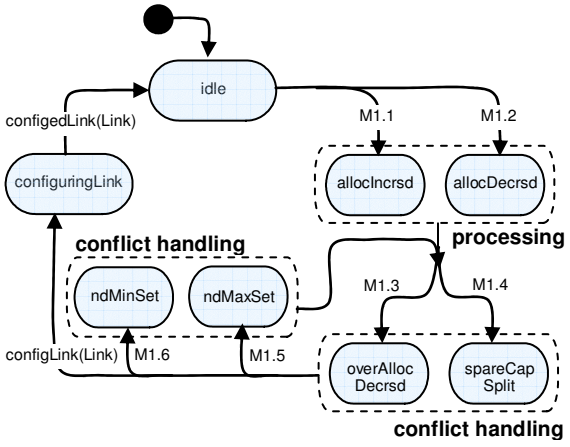

Figure 5. Main component behaviour with resolution logic.

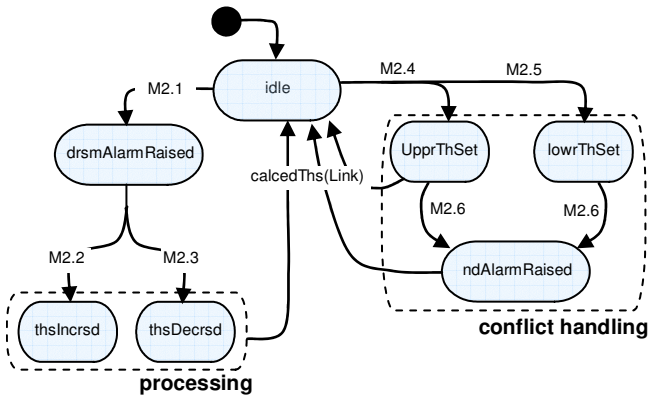

Figure 6. Monitoring component behaviour with resolution logic.

The resolution of the above conflicts implies that there will be under-allocation in case of an *ndMaxViolation* and over-allocation in case of an *ndMinViolation*. This situation can be resolved by re-visiting the previous stage and enforcing methods `M1.3` or `M1.4` to reduce or distribute the BW among the PHBs sharing the link, excluding the one involved in the violation.

It should be noted that conflict detection is triggered based on the state of the main component. While resolution is performed, the monitoring component calculates new thresholds and returns to the *idle* state (Fig. 6). If an *ndViolation* conflict is detected during the operation of the main component, further *resolution policies* should define how thresholds are to be treated. The actions to handle the event of an *ndMaxViolation* conflict could be to set the value of the upper threshold equal to the upper bound defined by the ND-refined policy (`M2.4`), and for an *ndMinViolation* to set the lower threshold to zero (`M2.5`) as to avoid further decrease in allocation. In both occasions an alarm is issued (`M2.6`) 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 and the DRsM modules involved.

```
setThUppr(Link, PHB, BW)                          (M2.4)
setThLowr(Link, PHB, BW)                          (M2.5)
raiseNDAlarm(ConflictType, Link, PHB, DRsMID)     (M2.6)
```

The detection for a *thIncompat* conflict is triggered once the main component enters the *configuringLink* state. In a similar manner to the strategies proposed above, this conflict is resolved by setting the upper threshold equal to the allocated value (`M2.4`). The latter need not be specified in the resolution action at specification-time as it can be acquired from the relevant parameter of the conflict predicate.

C. System Architecture

The system architecture for the process of conflict analysis is presented in Fig. 7, involving a centralised Policy Management Tool (PMT) and one instance of a DRsM module along with the associated PMA. Our approach towards static conflict detection as presented in [3], is based on the output of the refinement process, where high-level policy specifications introduced in the Policy Creation Environment (PCE) are decomposed into low-level implementable ones and mapped onto their respective EC representation. With domain information regarding mutually exclusive actions as described in [3], static detection logic is applied to a pool of low-level policies in the PMT, to determine if there are any domain-independent conflicts between them. Conflict-free policies are stored in a repository. Note that resolution policies are also checked for static conflicts; they are passed directly to the detection component after being mapped to EC representation. The communication between static 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

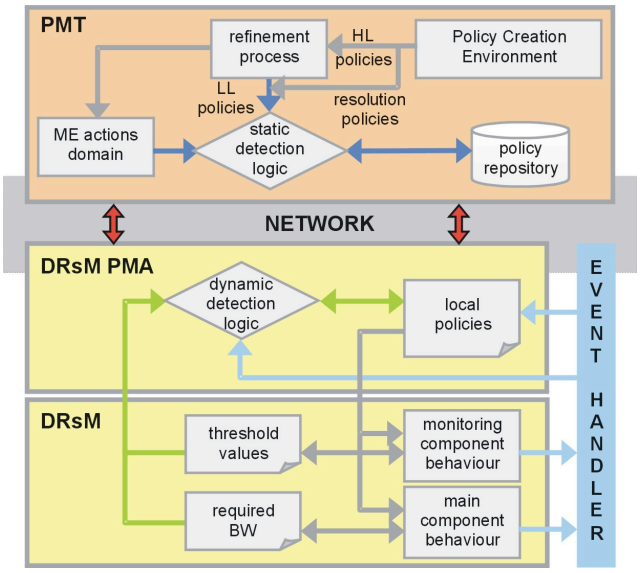

Figure 7. System architecture.

output of the refinement process, but also between new policies and ones already stored in the repository.

Policies related to a specific DRsM module are stored locally in the associated PMA, and then activated by relevant events from the two DRsM components or, in the case of resolution polices, by the detection component. As mentioned previously, dynamic detection logic is triggered with events generated by the main component. For example, the necessary logic to detect a possible *ndMaxViolation* conflict is activated once the allocation of a PHB has been decreased when resolving an *overAlloc* conflict. Once possible conflicting policies have been identified, a conflict will occur if there are inconsistencies related to the calculated values from the two DRsM components. These values are generated at each of the processing stages of the two components and are stored locally in the form of logic terms:

```
ths(Link, PHB, ThUppr, ThLowr)
reqBW(Link, PHB, BW)
```

The first concerns upper and lower threshold values from the monitoring component and the second, *required* BW values from the main component. Multiple instances of the above terms define values for the various PHBs sharing the links in the underlying network. The detection of a conflict triggers the appropriate resolution policy, which is enforced on the relevant DRsM component signalling the resolution of the conflict.

VI. 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 are taken from our initial implementation of a tool that supports dynamic conflict analysis. We assume that two traffic types are defined for the underlying network, namely EF and AF1, for which the associated values regarding

allocation, thresholds and ND constraints on link1 are presented in Table I. All values are expressed as a percentage of the total link capacity.

TABLE I. PHB ASSOCIATED VALUES

Link	PHB	Alloc	NDMin	NDMax	UpprTh	LowrTh
link1	EF	60	40	65	57	47
link1	AF1	40	20	50	32	24

Below, we define 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 ND-refined directives, p7 signifies the full allocation of link capacity, and policies p8-p11 provide resolution strategies for *underAlloc*, *overAlloc* and *ndMaxViolation* conflicts.

```
initiates(sysEvent(drsmAlarmRaised(upprTh, link1, ef)),
          oblig(p1, drsmPMA, op(mainMO,
                incrAllocRel(link1, ef, 20))), T).

initiates(sysEvent(drsmAlarmRaised(upprTh, link1, ef)),
          oblig(p2, drsmPMA, op(monitorMO,
                incrThsRel(link1, ef, 15))), T).

initiates(sysEvent(polReceived),
          oblig(p3, drsmPMA, op(mainMO,
                setBWMax(link1, ef, 65))), T).

initiates(sysEvent(polReceived),
          oblig(p4, drsmPMA, op(mainMO,
                setBWMin(link1, ef, 40))), T).

initiates(sysEvent(polReceived),
          oblig(p5, drsmPMA, op(mainMO,
                setBWMax(link1, ef, 50))), T).

initiates(sysEvent(polReceived),
          oblig(p6, drsmPMA, op(mainMO,
                setBWMin(link1, ef, 20))), T).

initiates(sysEvent(polReceived),
          oblig(p7, drsmPMA, op(mainMO,
                resAlloc(link1, 100))), T).

initiates(sysEvent(conflDetected(overAlloc,
                  conflictData(PolA, PolB, Link, ef))),
          oblig(p8, drsmPMA, op(mainMO,
                redOverBWEqual(Link))), T).

initiates(sysEvent(conflDetected(underAlloc,
                  conflictData(PolA, PolB, Link, ef))),
          oblig(p9, drsmPMA, op(mainMO,
                allocSpareBWExpl(Link, af, 100))), T).

initiates(sysEvent(conflDetected(ndMax,
                  conflictData(PolA, PolB, Link, PHB,
                               NDMaxBW))),
          oblig(p10, drsmPMA, op(mainMO,
                setNDMax(Link, PHB, NDMaxBW))), T).

initiates(sysEvent(conflDetected(ndMax,
                  conflictData(PolA, PolB, Link, PHB,
                               NDMaxBW))),
          oblig(p11, drsmPMA, op(monitorMO,
                setThUppr(Link, PHB, NDMaxBW))), T).
```

By using one of the conflict fluents (e.g. overAllocConflict) as a goal state of a deductive query, it is possible to detect any conflicts between active policies. The query has the following format, and is triggered at specific stages of the main component operation depending on the conflict type to be detected:

```
holdsAt(conflict(Type, ConflictData), T).
```

The results of the query indicate if there is a conflict of a particular type and the detection of a conflict causes the system to generate an event containing the conflict information which in turn triggers the relevant *resolution policy*. The following timeline shows the sequence of events (sysEvent(…)), 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 link1:

```
T - Event / Fluent
-------------------------------------------------------------------
 1 - sysEvent(drsmAlarmRaised(upprTh, link1, ef))
   |
 2 - oblig(p1, drsmPMA, op(mainMO,
   |                       incrAllocRel(link1, ef, 20)))
   | oblig(p2, drsmPMA, op(monitorMO,
   |                       incrThsRel(link1, ef, 15)))
 3 - doAction(drsmPMA, op(mainMO,
   |                       incrAllocRel(link1, ef, 20)))
   | doAction(drsmPMA, op(monitorMO,
   |                       incrThsRel(link1, ef, 15)))
   | sysEvent(allocIncrsd(link1, ef))
   | sysEvent(thsIncrsd(link1, ef))
   |
 5 - conflict(overAlloc, ConflictData)
   |
 6 - sysEvent(conflDetected(overAlloc,
   |                        conflictData(p1, p7, link1, ef)))
   |
 7 - oblig(p8, drsmPMA, op(mainMO, redOverBWEqual(link1)))
   |
 8 - sysEvent(doAction(drsmPMA, op(mainMO,
   |                               redOverBWEqual(link1)))
   |
 9 - sysEvent(overBWredced(link1))
   |
10 - conflict(NDviolation, ConflictData)
   |
11 - sysEvent(conflDetected(ndMax,
   |                  conflictData(p1, p3, link1, ef, 65)))
   |
12 - oblig(p10, drsmPMA, op(mainMO,
   |                       setNDMax(link1, ef, 65)))
   | oblig(p11, drsmPMA, op(monitorMO,
   |                       setThUppr(link1, ef, 65)))
   |
13 - sysEvent(doAction(drsmPMA, op(mainMO,
   |                               setNDMax(link1, ef, 65)))
   | sysEvent(doAction(drsmPMA, op(monitorMO,
   |                               setThUppr(link1, ef, 65)))
   |
14 - sysEvent(ndMaxSet(link1, ef, 65))
   | sysEvent(upprThSet(link1, ef, 65))
   |
15 - conflict(underAlloc, ConflictData)
   |
16 - sysEvent(conflDetected(underAlloc,
   |                        conflictData(p1, p7, link1, ef)))
   |
17 - oblig(p9, drsmPMA, op(mainMO,
   |                       allocSpareBWExpl(link1, af, 100)))
   | sysEvent(doAction(drsmPMA, op(mainMO,
   |                               allocSpareBWExpl(link1,
   |                               af, 100)))
   |
18 - sysEvent(underBWallocd(link1))
   |
19 - conflict(NDviolation, ConflictData)
   |
20 - oblig(drsmPMA, op(mainMO, configLink(link1)))
   |
21 - sysEvent(doAction(drsmPMA, op(mainMO,
   |                               configLink(link1)))
   |
22 - conflict(ThIncompat, ConflictData)
   |
   v
```

The generated alarm, at T=1, triggers policies p1 and p2, which increase the allocation and thresholds for EF traffic by 20% and 15% respectively. At this point, dynamic detection logic is triggered aiming to detect an allocation conflict, based on the newly calculated *required* BW value. The query signals

an *overAlloc* conflict between `p1` and `p7` since the sum of the required BW values for EF and AF1 exceed the maximum link capacity (112%). This result acts as a trigger for the relevant *resolution policy* (`p8`), which in turn reduces the over-allocated BW equally between the two PHBs, giving a new *required* BW of 66% and 34% for EF and AF1 respectively. The new value for EF traffic fulfils the conditions for an *ndMaxViolation* conflict, which is detected by the next query. The resolution of this conflict is handled by `p10` and `p11`, which set the required BW and upper threshold for EF traffic to 65%. This means that 1% of the link capacity remains unallocated, signalling an *underAlloc* conflict in the next query. At this point, the last of the resolution policies (`p9`) allocates the spare BW to AF1. Subsequent queries provide no solutions with respect to our conflict fluents, so the calculated values are configured accordingly.

VII. 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 [1] classify conflicts as domain-independent and application-specific, and in [9] the authors identify application-specific conflicts like conflicts of duty, conflicts of priorities for resources and self-management conflicts. The methodology presented in [1], and probably the only approach for conflict resolution in the literature, makes use of policy precedence rules to define which of the conflicting policies is to prevail after a conflict has been detected. The disadvantage of such a methodology is that, for dynamic conflicts the process could introduce delays while deciding for the appropriate resolution.

Work on computational efficiency for conflict detection and resolution mechanisms was presented in [11] and [2]. The authors identified several conflicts that may occur in open distributed systems and classified them into static and dynamic. Their detection mechanism involves identifying and predicting all possible conflicts at compile-time, based on knowledge of the temporal characteristics of the policies in the specification. In the case of dynamic conflicts the relevant conditions are stored in a database and subsequent monitoring of system events can lead to determining the occurrence of a conflict. Furthermore, they developed an approach as to *when* it is appropriate to resolve conflicts. Based on the fact that a resolution process can be computationally intensive, they proposed different approaches according to the likelihood of a conflict occurring and the cost of resolving that conflict. The actual resolution methodology presented by the authors follows the guidelines provided in [1], where policy precedence rules are being used.

The authors in [12] and [13] also make use of priorities when resolving a conflict. This is part of a *ratification* process where new policies are approved before being committed in a system. They identify the primitive operations that can be used for policy ratification, namely *dominance check*, *potential conflict check*, *coverage check* and *consistent priority assignment*, and they provide the relevant algorithms to implement these operations. Although this work is independent of the policy model used in a system, it does not address inconsistencies that may arise in application-specific environments. In particular, the assignment of priorities to conflicting policies may not be a flexible solution to the problem of conflict resolution, as demonstrated in some of our examples where new policies need to be enforced.

There are few conflict analysis examples that target specific application domains. In [14], 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 alarm the administrator to make the necessary changes. Another example involves work on using policies for adaptation of mobile devices [15] and proposes EC as a suitable formalism for obligation policy specification. However, conflict detection using the notation is still under development.

VIII. Conclusions and Future Work

In this paper we have indicated the types of potential conflicts that may arise in the domain of dynamic resource management for QoS support. We identified that conflicts may occur between policies applied to a single module of the TEQUILA architecture (i.e. DRsM), which we term as intra-module conflicts, or between policies specified for different modules (ND-DRsM) as a result of the architecture's hierarchical relationship, termed as inter-module conflicts. We classified these conflicts into domain-independent and application-specific, and specified the conditions under which these conflicts may arise.

Event Calculus was used to analyse the policy specification by defining the rules for conflict detection, and the supported reasoning methods provided the means to not only identify a conflict but also provide an explanation as to how that conflict occurred. For domain-independent conflicts static detection at specification-time was adequate to identify any conflicts in the policy specification, but for the case of applications-specific conflicts dynamic detection was necessary as their occurrence depended on the run-time state of the DRsM components.

While the resolution of domain-independent conflicts requires the intervention of a human administrator, we demonstrated how the occurrence of application-specific conflicts could be handled in an automated manner through the use of pre-defined *resolution policies*. Although we provided possible resolution strategies for the identified dynamic conflict types, an administrator may decide on other resolutions that suit the underlying network (e.g. topology) and traffic types. Furthermore, only few *resolution policies* are required per conflict type, catering for the different PHBs and associated

links. The fact that these policies can be specified in advance makes the resolution process efficient, as it is only a matter of triggering the appropriate decision instead of generating one on the fly.

Part of our future work will involve the classification, detection and resolution of possible conflicts related to the rest of the TEQUILA modules, such as the SLS Subscription and SLS Invocation. More specifically, we are interested in the hierarchical relationships between the modules that will possibly bring more inter-module related conflicts into the picture. This will allow us to identify conflicts originating from policies specified for higher-level modules and to possibly integrate the approach on conflict analysis with the ongoing work on policy refinement.

Another area that will be targeted is the one of computational efficiency. As far as static conflicts are concerned, there are no performance issues associated with their detection and resolution since this process is carried out off-line at policy specification-time. In the case of dynamic conflicts we need to consider the delay induced by the detection and resolution mechanism before providing a new configuration for the network.

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] N. Dunlop, J. Indulska, and K. Raymond, "Methods for conflict resolution in policy-based management systems," proceedings of the 7th International Conference on Enterprise Distributed Object Computing (EDOC 2003), Brisbane, Australia, 2003.

[3] M. Charalambides et al., "Policy conflict analysis for quality of service management," presented at 6th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2005), Stockholm, Sweden, 2005.

[4] P. Flegkas, P. Trimintzios, and G. Pavlou, "A policy-based quality of service management architecture for IP DiffServ networks," in IEEE Network Magazine Special Issue on Policy Based Networking, vol. 16 No. 2, pp. 50-56, 2002.

[5] 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.

[6] N. Damianou, N. Dulay, E.C. Lupu, and M.S. Sloman, "The Ponder policy specification language," presented at 2nd IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2001), Bristol, UK, 2001.

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

[8] 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.

[9] J.D. Moffett and M.S. Sloman, "Policy conflict analysis in distributed system management," Journal of Organisational Computing, vol. 4, pp. 1-22, 1994.

[10] A.K. Bandara et al., "Policy refinement for DiffServ quality of service management," proceedings of 9th IEEE/IFIP Integrated Management Symposium (IM 2005), Nice, France, 2005.

[11] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," proceedings of the 6th International Conference on Enterprise Distributed Object Computing (EDOC 2002), Lausanne, Switzerland, 2002.

[12] D. Agrawal, J. Giles, K.W. Lee, and J. Lobo, "Policy ratification," presented at 6th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2005), Stockholm, Sweden, 2005.

[13] D. Agrawal, J. Giles, K.W. Lee, and J. Lobo, "Policy-based management of networked computing systems," in IEEE Communications Magazine, vol. 43 No. 10, pp. 69-75, 2005.

[14] 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.

[15] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst, "Utilising the Event Calculus for policy driven adaptation on mobile systems," presented at 3rd IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2002), Monterey, CA, USA, 2002.