Policy Refinement for DiffServ Quality Of Service Management

A. Bandara, E. Lupu, A. Russo,
N. Dulay, M. Sloman
Imperial College London,
180 Queensgate, London SW7 2AZ, UK
{a.k.bandara, e.c.lupu, a.russo, n.dulay, m.sloman}@imperial.ac.uk

P. Flegkas, M. Charalambides,
G. Pavlou
CCSR, University of Surrey,
Guildford, GU2 7XH, UK
{p.flegkas, m.charalambides, g.pavlou}@eim.surrey.ac.uk

Abstract:

Policy-based management provides the ability to dynamically re-configure DiffServ networks such that desired Quality of Service (QoS) goals are achieved. This includes network provisioning decisions, performing admission control, and adapting bandwidth allocation dynamically. QoS management aims to satisfy the Service Level Agreements (SLAs) contracted by the provider and therefore QoS policies are derived from SLA specifications and the provider's business goals. This *policy refinement* is usually performed manually with no means of verifying that the policies written are supported by the network devices and actually achieve the desired QoS goals. Tool support is lacking and policy refinement has rarely been addressed in the literature. This paper extends our previous approach to policy refinement and shows how to apply it to the domain of DiffServ QoS management. We make use of goal elaboration and abductive reasoning to derive strategies that will achieve a given high-level goal. By combining these strategies with events and constraints, we show how policies can be refined, and what tool support can be provided for the refinement process using examples from the QoS management domain. However, the approach presented here can be used in other application domains such as storage area networks or security management.

Keywords

Policy derivation, Goal elaboration, Refinement patterns

1. Introduction

Network Quality of Service (QoS) management requires administrators to manage the network devices and infrastructure to achieve predictable performance. The Differentiated Services (DiffServ) architecture [1] can achieve this by aggregating network traffic into defined classes of service, and configuring routers to treat each of these classes appropriately. This results in a network where, at each hop, a packet might be handled differently based on the DiffServ class it belongs to. Policy-based management provides the ability to dynamically configure a system, by separating the rules that govern a system's behaviour from the functionality supported by it. Policies can be specified, and applied to large numbers of devices uniformly. In DiffServ, policies can be used to dynamically reconfigure routers such that the desired QoS goals are achieved as well as to perform admission control. It is important to be able to analyse policies to ensure consistency and to ensure that key

properties are preserved in the network configuration, e.g. traffic marked in the same way is not allocated to different queues. Although adaptation can be realised through general scripting languages, policy languages adopt a more succinct, declarative, form in order to facilitate analysis. Although many policy languages have been proposed, policy analysis techniques remain poorly explored. Unless such techniques are developed and used in network management and provisioning tools, the additional expense required to deploy policy-based management will remain difficult to justify.

The SLAs which have to be satisfied by the network, as well as the derived QoS policies required to satisfy the SLAs, will change frequently. This process of deriving policies from the SLA specifications is recognized as one of the most difficult research challenges and is not fully automatable. However, tool support to assist administrators in the refinement of policies would significantly reduce and improve network administration tasks especially when combined with analysis tools to ensure that only consistent specifications are derived. Although we contend that policy refinement cannot be automated in general, techniques and tool support for refinement can be developed. Moreover, increasing support can be achieved when constraining the problem to a well defined functional area, such as QoS management, where application specific knowledge can be encoded and used.

Policy refinement is the process of transforming a high-level, abstract policy specification into a low-level, concrete one. Moffett and Sloman [2], identify the main objectives of a policy refinement process as:

i) Determine the resources that are needed to satisfy the requirements of the policy.
ii) Translate high-level policies into enforceable, operational policies.
iii) Verify that lower level policies actually meet the high-level policy requirements.

(i) involves mapping abstract entities defined as part of a high-level policy to concrete objects/devices that make up the underlying system. (ii) specifies the need to ensure that derived operational policies are in terms of operations supported by the underlying system. (iii) requires a process for incrementally decomposing abstract requirements into more concrete ones, ensuring that at each stage the decomposition is correct and consistent. In previous work [3], we have proposed an approach that meets these objectives by elaborating high-level, abstract goals into more concrete ones and using abductive reasoning to derive the actions (strategies) that are supported by the system for achieving these concrete goals. We show how these strategies can then be used in specifying policies that can be enforced by the system to achieve the original goal. The goal elaboration technique makes use of the KAOS [4] requirements engineering approach, which is based on the use of formal specifications in conjunction with policy refinement patterns that are proved to be correct. The most useful patterns are likely to be application-specific, e.g. for QoS management, storage or security, although KAOS also defines a set of application-independent patterns together with proofs of their correctness.

This paper focuses on the application of our refinement technique to DiffServ QoS management. By limiting the scope to this application domain, we are able to specify application specific refinement patterns and maximise the level of automation in the refinement process. In order to identify the goals, strategies and policies

involved in DiffServ QoS management we use the framework developed in the context of the EU IST TEQUILA project [5]. TEQUILA uses DiffServ, with Multi-Protocol Labelled Switching (MPLS) [6] to provide dynamic adaptation to varying traffic requirements. This adaptation is performed using a combination of online and offline techniques – from network dimensioning calculations that determine the upper/lower bounds of network parameters based on the Service Level Agreements (SLAs) and traffic forecasts; to dynamic resource and route management modules that make real-time changes to the router configuration to handle variations in traffic.

In the next section we present background information about the TEQUILA framework and our policy refinement approach. Section 3 presents example scenarios from the QoS management domain, and identifies the goals, strategies and policies involved. In section 4 we show how the examples can be generalised into application-specific refinement patterns and later reused in new situations. Section 5 presents related work; and section 6 discusses our conclusions and future work.

2. Background

2.1 Approach to Policy Refinement

The first phase of the policy refinement process is a technique for refining high-level goals into concrete achievable goals, often referred to as System Requirements. The next phase of the refinement process maps these system requirements to specific modules/operations that are available within the system. In this process, each high-level goal is refined into sub-goals, forming a refinement hierarchy where the dependencies between goals at different levels of refinement are based on the type of goal decomposition used (AND/OR). Additionally there can be dependencies between goals in different hierarchies. The refinement process involves following a particular path down the hierarchy, at each stage verifying the feasibility of achieving the higher-level goal in terms of the lower-level ones. If it is discovered that a high-level goal cannot be achieved, then we have to either manually decompose the goal, such that suitable lower-level goals can be derived, or increase the system's functionality by adding additional management procedures and services.

KAOS [4] is a formal technique for goal elaboration, where each goal is represented as a Temporal Logic rule and refinement patterns are used to decompose the original goal into a logically entailed set of sub-goals. This process results in a set of refined goals, and the identification of objects and operations that might operationalise those goals. Whilst KAOS does not provide automated support for goal refinement, it does define a library of domain-independent refinement patterns that have been logically proved correct. The following table shows some patterns of AND-decomposition for goals of the form $P \Rightarrow \Diamond Q$ (if P holds, then Q will eventually hold in the future):

Table 1: Selection of Domain-independent goal elaboration patterns

Ref	Goal	Subgoals
GP1	$P \Rightarrow \Diamond Q$	$(P \Rightarrow \Diamond R) \wedge (R \Rightarrow \Diamond Q)$
GP2	$P \Rightarrow \Diamond Q$	$(P1 \Rightarrow \Diamond Q) \wedge (P2 \Rightarrow \Diamond Q) \wedge (P \Rightarrow P1 \vee P2)$
GP2'	$P \Rightarrow \Diamond Q$	$(P \Rightarrow P1 \wedge P1 \Rightarrow \Diamond Q) \vee (P \Rightarrow P2 \wedge P2 \Rightarrow \Diamond Q) \vee (P \Rightarrow P2 \wedge P2 \Rightarrow \Diamond Q)$

Having refined the abstract goals into lower-level ones, the next phase of the process is to assign each refined goal to a specific object/operation such that the final system will meet the original requirements. Since KAOS does not provide support for automating this, we propose the following method for inferring the mechanism by which the system can achieve a given goal.

At a given level of abstraction there will be some description of the system (SD) and the goals (G) to be achieved by the system. The relationship between the system description and the goals is the Strategy (S), i.e. the Strategy describes the mechanism by which the system represented by SD achieves the goals denoted by G. Formally this would be stated as:

```
        (1) - SD, S ⊢ G
```

This requires a representation of the system description, in terms of the properties and behaviour of the components, together with a definition of the goals that the system must satisfy. We use Statecharts to describe system behaviour, where each transition indicates the invocation of an operation and/or the occurrence of a system event that can trigger the transition. Guards are specified for transitions where there are some pre-conditions for invoking the operation. We have chosen Statecharts for two reasons: first, because it is unrealistic to consider that users will provide system descriptions in the underlying formal specification language whereas Statecharts are a well-known design level behavioural specification notation and second, because it is possible to translate from the Statechart specification to the underlying formalism. We use Event Calculus (EC) [7] as the underlying formalism for analysis and refinement. The mapping between the system descriptions and policy language to their EC representation is detailed in [3]. Using the EC representation of the system, and given the relationship between the system description, strategy and goal defined in (1) above, we then use *abduction* to programmatically infer the strategies that will achieve a particular goal. Given the rules describing a system (SD) and the definition of some desired system state (i.e., the goal - G), abduction allows us to derive the facts that must be true for the desired system state to be achieved. As the goal is represented by a desired system state abduction is essentially deriving a path in the statechart from some initial state to the desired one. This path is the derived strategy and can be represented using the following syntax:

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

Whether a strategy should be encoded as policy, or as system functionality, will depend on the particular application domain. Although there is no obvious way to automate this decision, we propose the following guidelines to identify the situations where a policy-based implementation would be appropriate:

1. If the goal refinement results in a disjunction of sub-goals (i.e. the high-level goal can be achieved by one of an OR-decomposed set of sub-goals), the strategies derived for each of the sub-goals could be encoded as policies.
2. If the system supports multiple strategies for achieving a given goal, each of these strategies could be encoded in a separate policy. This situation might arise when the abductive process yields multiple solutions.

3. If a strategy has parameter values that may need to change in the future, implementing the strategy in a policy will provide the flexibility to do this.

In addition to elaborating goals and deriving strategies, it is necessary to map abstract entities to concrete objects/devices in the system. For example, there might be an abstract "Network" entity that logically consists of "Routers", "Links" etc., each consisting of the relevant managed objects. A domain hierarchy is used to represent the relationships between the various abstract entities and the low-level concrete objects [8]. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in large systems according to geographical boundaries, object type, responsibility and authority. Membership in a domain is explicit and not defined in terms of a predicate on object attributes. An advantage of specifying policies in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies.

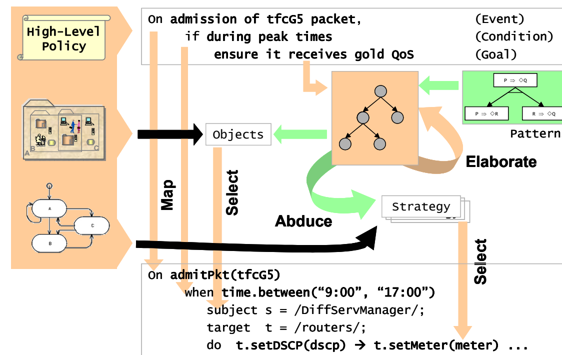

Figure 1: Policy Refinement Process

Combining this domain hierarchy based approach for refining abstract entities with the goal elaboration and strategy derivation techniques, the overall policy refinement process can be summarized as follows. The user provides information about the system behaviour, in the form of statecharts, together with the domain hierarchy for the managed objects and the high-level policy they are interested in refining. This policy would be of the form "On ***event***, if ***condition*** holds then ***achieve goal***". The KAOS approach is applied to elaborate the high level policy, making use of both domain-independent and domain-specific refinement patterns provided by the system. At each stage of elaboration, the system description and the goals are used to attempt to abduce a strategy for achieving the goal. If no strategy can be derived, then either the goals are elaborated further, or the system description is augmented with more detail. Once a strategy is identified, it is used in the action clause of the final policy. The domain hierarchy is used to identify the exact objects in the system that correspond to those entities mentioned in the high-level policy which are used in the subject and target clauses of the final policy. Finally the event and constraints of the high-level policy are mapped, by the user, into the final policy (Figure 1). This

final step is a manual one since there is no easy way to capture the domain information necessary for translating high-level events and constraints into lower-level ones. This is not a major disadvantage, as, these mappings can be done once and encoded into application specific refinement patterns that are reusable.

Automating this technique requires tools that allow users to specify the system behaviour and goal information in a high-level notation, such as UML, and then translate this representation into Event Calculus for analysis. Also, the results of the analysis should be presented in an easy to understand form. To achieve this, we are developing a tool that integrates a UML editor (ArgoUML), with an abductive reasoning engine (A-System with SICStus Prolog [9]). The same abductive reasoning framework has also been used to develop the policy analysis approach presented in [10]. Finally, this refinement and analysis tool is being integrated with the Ponder policy system [8]. The translation between Ponder policies and their Event Calculus representation has been presented in [10].

2.2 TEQUILA DiffServ Framework

The TEQUILA framework operates in two modes – an offline mode that determines the configuration required to meet long-term traffic demands; and a run-time mode that adapts the configuration to meet short-term traffic variations. It can be decomposed into three 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 network configuration. The Monitoring subsystem provides the above systems with the appropriate network measurements and assures that the contracted SLSs are indeed delivered at their specified QoS. Figure 2 shows a logical representation of this architecture. The TEQUILA framework has been previously presented [11, 12], so we describe here only the behaviour of the SLS-S and DRsM components which are used in the scenarios presented in the next section.

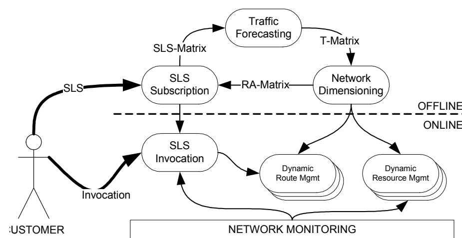

Figure 2: TEQUILA DiffServ Architecture

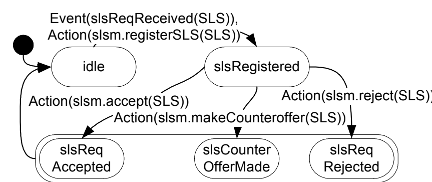

Figure 3: SLS Subscription Module Behaviour

The SLS-S module (Figure 3) performs admission control, calculates counter-offers and updates of traffic forecasts. Admission control is controlled by policies and so is the most relevant component for policy refinement. The SLS-S module uses the parameters of each requested SLS to calculate the expected traffic load based on

traffic demand forecasts. This traffic is then aggregated with the expected traffic accumulated from the SLSs established during this Resource Provisioning Cycle (RPC). The resulting aggregated traffic defines the maximum potential demand and is mapped against the corresponding entries of the resource availability matrix (RA-Matrix). The result of this mapping is used by the admission control algorithm, when deciding whether requests should be accepted or rejected. Requests are rejected if the risk is too high of overwhelming the network with traffic that cannot be served with the guaranteed QoS. This is shown in Figure 3. A more detailed description of the subscription admission control algorithm can also be found in [12].

Traffic Engineering comprises 3 functional blocks. Network Dimensioning (ND) performs the long-to-medium term network configuration and is responsible for mapping the traffic onto the physical network resources to accommodate forecasted traffic demands. The output of ND is fed to Dynamic Route Management (DRtM), Dynamic Resource Management (DRsM), and to SLS-S in order to provide the traffic limits on which admission control decisions for future SLS subscriptions are based. The DRtM is distributed, operates at each edge router and manages the routing processes according to the guidelines produced by ND. The DRsM is also distributed, with an instance present at each router interface and ensures that link capacity is appropriately distributed between the PHBs sharing the link. This is achieved by configuring buffer and scheduling parameters according to ND directives, and taking into account the actual experienced load.

The DRsM can be further decomposed into two components. The first monitors PHB utilization and raises threshold-crossing alarms when the bandwidth consumed by a PHB exceeds an upper threshold or drops below a lower threshold. In fact, two values could be used for each threshold (trigger and clear values) to avoid repeated alarms when small oscillations occur. Once an alarm is raised, the DRsM calculates a new bandwidth allocation and configuring the link appropriately; or triggers a new resource provisioning cycle if sufficient bandwidth cannot be allocated. Policies determine how to calculate the new value, configure the link or trigger a new RPC. Figure 4 shows the behaviour of the DRsM components.

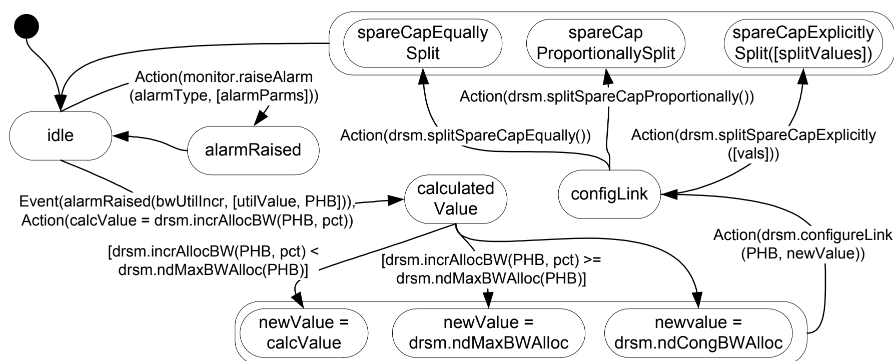

Figure 4: Dynamic Resource Management Component Behaviour

3. DiffServ Goals, Strategies and Policies

We present here two TEQUILA scenarios. The first describes the admission control performed when customers register new Service Level Agreements; and the second describes the response to a short-term increase in the traffic from a particular customer. For each scenario we present the goals, strategies and policies that apply.

3. 1 Scenario 1: New SLS Subscription

Consider an example where a new SLS from the customer, AOL, requires a pipe between routers R1 and R6 with Expedited Forwarding (EF) per hop behaviour, 20ms delay, zero packet loss, and a 10Mbps throughput guarantee. **SLS[***customer*: aol; *scope*: pipe(r1,r6); *qos*: qosClass(EF, 20, 0); *bwReq*: bw(10Mbps)**]**, is presented to the SLS-S subscription module. The SLS-S module registers the SLS, compares its contents with the RA-Matrix and decides whether to accept, reject or make a counteroffer. Policies are used to influence the choice of the SLS-S module. The policy that applies depends on the goals that need to be achieved. For example, the highest level goal below ensures that the SLS request is processed:

```
G1: Goal SLSRequestProcessed
        FormalDef  slsReqReceived(SLS) ⇒ ◇ slsRequestProcessed(SLS).
```

Since applying the abductive analysis to the system description of the SLS-S module does not produce strategies for achieving this goal, it is necessary to elaborate it further by the domain-independent pattern `GP2'` (see Table 1) to decompose the above goal into the following sub-goals. In each case we use abduction to derive a strategy:

```
G2: Goal SLSRequestAccepted
        FormalDef slsReqReceived(SLS) ⇒ slsReqAccepted(SLS) ∧
                  slsReqAccepted(SLS) ⇒ ◇ slsRequestProcessed(SLS).

G3: Goal SLSRequestRejected
        FormalDef slsReqReceived(SLS) ⇒ slsReqRejected(SLS) ∧
                  slsReqRejected(SLS) ⇒ ◇ slsRequestProcessed(SLS).

G4: Goal SLSCounterofferMade
        FormalDef slsReqReceived(SLS) ⇒ slsCounterofferMade(SLS) ∧
             slsCounterofferMade(SLS) ⇒ ◇ slsRequestProcessed(SLS).

S1: Strategy G2: SLSRequestAccepted
        OnEvent         slsReqReceived(SLS)
        DerivedActions  slsm.registerSLS(SLS) -> slsm.accept(SLS).

S2: Strategy G3: SLSRequestRejected
        OnEvent         slsReqReceived(SLS)
        DerivedActions  slsm.registerSLS(SLS) -> slsm.reject(SLS).

S3: Strategy G4: SLSCounterofferMade
        OnEvent         slsReqReceived(SLS)
        DerivedActions  slsm.registerSLS(SLS) -> slsm.makeCounteroffer(SLS).
```

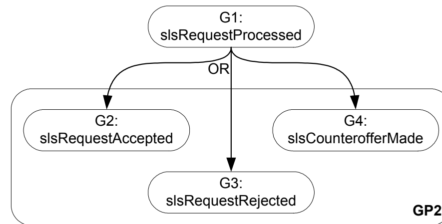

Figure 5: Goal decomposition for SLS subscription scenario

As shown in Figure 5, goal elaboration yields a disjunction of goals (G2-G4), and the user can select the sub-goal that best satisfies the requirement. Strategies (S1-S3) are derived automatically and identify the action sequences (->, sequence operator) that achieve each of the sub-goals. In this scenario, the required high-level policy is that SLS requests from customer 'AOL' with qosClass(EF, 20, 0) should be accepted if the bandwidth requested is less than the bandwidth available in the RA-Matrix for the same QoS class. As this policy achieves the `SLSRequestAccepted` goal we can encode the corresponding strategy into a policy as follows:

```
P1: inst oblig /policies/slsm/acceptAOLSLS_P1 {
        on    slsReqReceived(SLS);
        subj  s = /slsmPMA;
        targ  t = s.slsm;
        do    t.register(SLS) -> t.accept(SLS);
        when  SLS.customer = 'aol' && SLS.qosClass = qosClass(ef, 20, 0) &&
               t.getAvailBW(SLS.qosClass) > SLS.bwReq;
    }
```

Whilst the strategy is derived automatically, user intervention is required to map the event and constraints specified in the goal into the policy. Additionally, the system helps the user select the specific subjects and targets by automatically identifying objects of the required types in the domain hierarchy. Thus, the high-level goal specified by the network administrator is refined into a concrete policy.

3. 2 Scenario 2: Increase in traffic

This scenario illustrates how the TEQUILA framework responds to short-term traffic changes from customers. The network administrator wants to ensure that when such an increase occurs during between 11am and 1pm and causes a network utilisation greater than 85% of the maximum allocation calculated by the ND module, the bandwidth allocation should be increased by 10% and spare capacity should be equally split amongst the PHBs. In this situation the Dynamic Resource Management (DRsM) module at each link along the traffic route would respond as follows:

1. On receiving a traffic increase alarm, the DRsM decides on the appropriate action to adapt to the increase using guideline values for maximum, minimum and congestion bandwidth allocations provided by the ND.
2. Configure the link/PHB with this new value and decide on how to allocate any spare link capacity amongst all the link/PHBs

Policies are used at each of the stages above, to decide how to calculate the new bandwidth allocation, and how to distribute spare link capacity. In each case the exact policy to be used depends on the required goal. For the policy decisions on calculating the new bandwidth allocation and then allocating spare capacity, the high-level goal (G6) is to achieve the state "adapted configuration" when an alarm is raised. This can be stated as follows:

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

In this case the abductive analysis of G6 yields no strategy, so the goal must be elaborated further. Applying `GP2'` yields the sub-goals `NewRPCRequested` (G7) or `CalculatedConfigNewBWAllocation` (G8). Each leads to the high-level goal G6 being satisfied as shown in their formal definitions below.

```
G7: Goal NewRPCRequested
     FormalDef  alarmRaised(bwUtilIncr, [utilValue, PHB]) ⇒ requestedNewRPC ∧
                requestedNewRPC ⇒ ◇ configAdapted.

G8: Goal CalculatedConfigNewBWAllocation
     FormalDef  alarmRaised(bwUtilIncr, [utilValue, PHB]) ⇒ calcAndConfigNewBWAlloc ∧
                calcAndConfigNewBWAlloc ⇒ ◇ configAdapted.
```

In the scenario, the high-level policy requires calculating and configuring a new bandwidth allocation, represented by goal G8 above. However, since it is not possible to automatically derive a strategy for this goal, it is necessary to elaborate it further, this time using a combination of the patterns `GP2'` and `GP1`. Figure 6 indicates the applicable patterns at each stage with the following goals:

```
 G9: Goal calcNewBWAlloc
     FormalDef  calcNewBWAlloc(newValue) ⇒ ◇ configNewBWAlloc.

G10: Goal configNewBWAlloc
     FormalDef  configNewBWAlloc ⇒ ◇ configAdapted.

G11: Goal setCalculatedNewBWAlloc
     FormalDef  calcNewBWAlloc (newValue) ⇒ (newValue = calcValue) ∧
                (newValue = calcValue) ⇒ ◇ configNewBWAlloc.
G12: Goal overrideNewBWAllocNDMax
     FormalDef  calcNewBWAlloc (newValue) ⇒ (newValue = drsm.ndMaxBWAlloc) ∧
                (newValue = drsm.ndMaxBWAlloc) ⇒ ◇ configNewBWAlloc.

G13: Goal overrideNewBWAllocNDCong
     FormalDef  calcNewBWAlloc (newValue) ⇒  (newValue = drsm. ndCongBWAlloc) ∧
                (newValue  = drsm.ndCongBWAlloc) ⇒ ◇ configNewBWAlloc.

G14: Goal propSplitSpareCapacity
     FormalDef  configNewBWAlloc ⇒ spareCapProportionallySplit ∧
                spareCapProportionallySplit ⇒ ◇ configAdapted.

G15: Goal equalSplitSpareCapacity
     FormalDef  configNewBWAlloc ⇒ spareCapEquallySplit ∧
                spareCapEquallySplit ⇒ ◇ configAdapted.

G16: Goal explicitySplitSpareCapacity
     FormalDef  configNewBWAlloc ⇒  spareCapExplicitlySplit([splitValues]) ∧
                spareCapExplicitlySplit([splitValues]) ⇒ ◇ configAdapted.
```

In this scenario, the goals of the administrator are `G11: setCalculatedNewBWAlloc` and `G15: equalSplitSpareCapacity`. So, we are interested in the strategies for setting the new bandwidth to the newly calculated value and splitting spare capacity equally. Performing the abductive analysis on the statechart representation of the DRsM calculation and configuration module behaviours yields the following strategy, which in turn can be encoded into a policy:

```
S5: Strategy G11: setCalculatedNewBWAlloc && G15: equalSplitSpareCapacity
     OnEvent         alarmRaised(bwUtilIncr, [utilValue, PHB])
     DerivedActions  calcValue = drsm.incrAllocBW(PHB, pct) ->
                     drsm.configureLink(PHB, calcValue) -> drsm.splitSpareCapEqually
     Constraints     drsm.incrAllocBW(PHB, pct)<drsm.ndMaxBWAlloc(PHB).

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

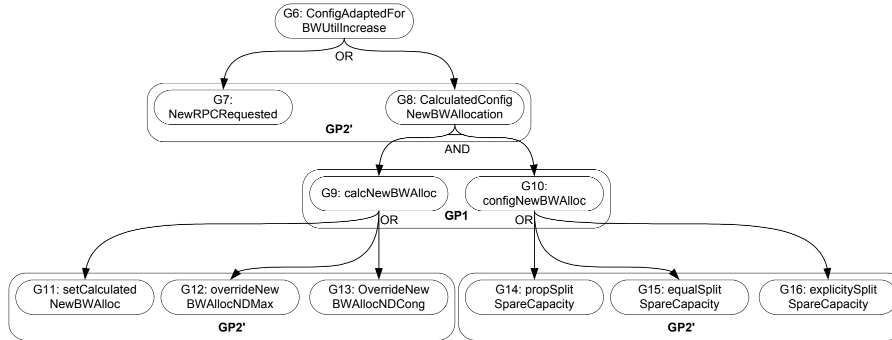

Figure 6: Goal decomposition for traffic increase scenario

Note that here, the abductive analysis results in a strategy that includes constraints. These are derived from the guards defined in the state chart of the system behaviour and must therefore be included in addition to any other constraints manually mapped from the high-level policy. This is illustrated in policy P3, which combines the strategy constraint with the time constraint from the high-level policy.

4. Application-specific Refinement Patterns

In the scenarios described above, specific policies were derived by refining individual goals. Refining every goal would be onerous for network administrators as the process is only partially automated. Therefore, it is useful to define refinement patterns that directly relate a goal, to the set of policies that could achieve it. Each pattern can also be parameterized according the specifics of the high-level goal. To achieve this, we introduce the following syntax for policy refinement patterns:

```
policyPattern patternName(ParameterList) {
  description       A description of the policy pattern.
  goalHierarchy goal [refinesTo (goalHierarchy)]
  policies {// Group of policies that will achieve the goal hierarchy for this pattern.}
}
```

The network administrator can use the derived strategies and policies in the above construct to capture the pattern for later reuse. For example, in scenario 1, where the high-level goal was to process SLS requests, we derived a policy that achieved the sub-goal that the SLS request was accepted when constraints relating to the customer, QoS class and available bandwidth were met. The administrator can generalise this policy by parameterising these constraint values and by using the `policyPattern` construct described above. The pattern for this situation is shown below in Figure 7.

The network administrator can achieve the same goal for a different customer or QoS class, by instantiating this pattern with the appropriate values. The policy management tool can aid the administrator to select the appropriate refinement pattern by providing a search interface for the pattern repository that matches the goals presented (including the constraints), with goals specified in the patterns. Note that the goal definition in the above example only mentions those goals which are

satisfied by the pattern; SLSRequestRejected and SLSCounterofferMade are omitted. This ensures that this pattern will only be highlighted when the administrator searches for patterns relating to SLSRequestAccepted.

```
policyPattern /ptn/acceptSLS(String customer, QoSClass qc) {

description   Accept incoming SLS from customer provided it is for a \
              specified QoSClass and bandwidth can be satisfied by available resources.

  goalHierarchy  SLSRequestProcessed  refinesTo (SLSRequestAccepted);

policies {
 oblig acceptSLS1 {
 on    slsReqReceived(SLS);
 subj  s = /slsmPMA;
 targ  t = s.slsm;
 do    t.register(SLS) -> t.accept(SLS);
 when  SLS.customer=customer && SLS.qosClass=qc && t.getAvailBW(SLS.qosClass)>SLS.bwReq; }}}
```

Figure 7: Example policy refinement pattern for SLS subscription

For example, to create policies that ensure that SLS requests from customer ‘pipex’ are accepted when they contain the QoS class qosClass(AF1, 50, 15%), the administrator would search for patterns relating to the SLSRequestAccepted goal. Having identified the above pattern, he would instantiate it as follows:

```
inst policyPattern acceptPipexSLS = /ptn/acceptSLS(‘pipex’, qosClass(AF1,50,15%));
```

The policy management system would then instantiate each of the policies in this pattern with the parameters specified. Once the policy has been instantiated, the overall policy specification can be analysed for inconsistencies as shown in [10].

5. Related Work

There are few practical studies on policy refinement. Power [13] is a policy-authoring environment where a domain expert specifies policy templates (as Prolog programs), which guide the user in selecting the elements from an information model to be included in the policy. This approach lacks any analysis capabilities to evaluate the consistency of the results. Additionally, Power does not provide support for automatically deriving the actions to be included in a policy. Therefore, domain experts must have a detailed understanding of system and formalism. Our refinement patterns are similar to the Power templates, however, our approach incorporates a complete analysis technique and provides automated derivation of action sequences.

Verma presents an approach to policy translation for DiffServ QoS management that is based on a set of tables which identify the relationships between Users, Applications, Servers, Routers and Classes of Service supported by the network [14]. When presented with new SLSs, the system performs a series of table look-ups to identify the correct configuration for the specified user, application and service class. This technique is fully automated, but depends on the correctness of the table which requires domain expertise. Furthermore, this approach is inflexible, as it supports only specific types of SLA and low-level device policies.

6. Conclusions and Future Work

This paper focuses on policy refinement for QoS management. Through specific examples, we have shown how goals can be elaborated using refinement patterns and how abduction can be used to derive strategies that achieve these goals. We have also shown how these strategies can be encoded into policies for specific scenarios and also in general refinement patterns for later reuse. Note that the techniques employed: goal elaboration, strategy derivation and use of refinement patterns are not QoS specific and can be used in other application domains.

Our refinement process is built on a systematic, formal and semi-automated approach to goal refinement thus ensuring that derived strategies meet the high-level policy requirements. System descriptions are used to ensure that derived policies are enforceable by the system. Using domain hierarchies to model the relationships between abstract entities and concrete objects, together with type information, permits identifying the objects required to execute strategies. These features illustrate how this solution satisfies the objectives of policy refinement identified in [2].

It is necessary to analyse policies to detect inconsistencies. After preliminary work on modality and application specific conflicts [15], we have shown how an Event Calculus representation of both policies and managed systems can be used, together with abductive reasoning for policy analysis [10]. Like the refinement technique presented here, the analysis uses a statechart representation of system behaviour and the domain hierarchy. The abduction process derives not only the presence of conflicts but also a description of the conditions under which the conflicts will occur. Since both the analysis and the refinement techniques are based on the same formalism the two can easily be integrated.

An important consideration when using formal techniques is to ensure that the implementation is decidable and computationally feasible. In our implementation, we ensured this by limiting ourselves to stratified logic programs. This permits a constrained use of recursion and negation while disallowing those combinations that lead to undecidable programs [16]. Stratified logic programs are a decidable class of first order logic [17, 18] and are decidable in polynomial time [18].

Although the underlying approach uses formal specifications, network operators need only use libraries of goals and refinement patterns together with high-level notations (e.g. Statecharts) for describing the managed system. Thus, the selection of goals and refinement patterns can be mostly driven by their natural language description. We are developing tools that minimise the amount of required knowledge and intervention from network operators.

One limitation of the work presented is that it does not permit calculating the parameter values for management operations such as the input rate of the DiffServ meters. We plan to investigate integrating constraint logic programming techniques to provide such capabilities. Another limitation is that we use goals decomposed using solely the AND/OR connectives ignoring any temporal ordering considerations. Whilst the time information provided by the Event Calculus may be used for this purpose, the complexity implications require further investigation.

We are currently developping an integrated analysis/refinement tool based on an abductive reasoning engine (A-System/Prolog) and using a UML and Policy editor for specification and user interaction. This effort is already well under way.

7. Acknowledgements

We acknowledge financial support for this work from the EPSRC (Grant Nos: GR/R31409/01 and GR/S79985/01), CISCO Systems Inc. and IBM Research.

8. References

[1] M. Carlson, et al., "An Architecture for Differentiated Services," in *Network Working Group - RFC2475, http://www.ietf.org/rfc/rfc2475.txt*, 1998.
[2] J. Moffett and M. S. Sloman, "Policy Hierarchies for Distributed Systems Management," *IEEE JSAC*, 11(9):1404-14, Special Issue on Network Management, 1993.
[3] A. K. Bandara, E. C. Lupu, and A. Russo, "A Goal-based Approach to Policy Refinement," *IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2004)*, IBM TJ Watson Research Centre, New York, USA, June, 2004.
[4] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," *4th ACM Symposium on the Foundations of Software Engineering (FSE4)* pp 179-190, 1996.
[5] P. Flegkas, P. Trimintzios, and G. Pavlou, "A Policy-based Quality of Service Management Architecture for IP DiffServ Networks," *IEEE Network*, 16(2):50-56, 2002.
[6] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," in *Network Working Group - RFC3031, http://www.ietf.org/rfc/rfc3031.txt*, 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] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman, "Tools for Domain-based Policy Management of Distributed Systems," *IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, Florence, Italy, 2002.
[9] B. van Nuffelen and A. Kakas, "A-System : Programming with abduction," presented at Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), 2001.
[10] A. K. Bandara, E. C. Lupu, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2003), Lake Como, Italy, 2003.
[11] P. Trimintzios, et al., "Service-driven Traffic Engineering for Intra-domain Quality of Service Management," *IEEE Network Magazine, 17(3):*29-36, 2003.
[12] E. Mykoniati, et al., "Admission Control for Providing QoS in IP DiffServ Networks: the TEQUILA Approach," *IEEE Communications Magazine*, 41(1), 2003.
[13] M. Casassa Mont, A. Baldwin, and C. Goh, "POWER Prototype: Towards Integrated Policy-Based Management," HP Laboratories Bristol, Bristol, UK October 1999.
[14] D. C. Verma, *Policy-Based Networking: Architecture and Algorithms*. New Riders, 2001.
[15] E. C. Lupu and M. S. Sloman, "Conflicts in Policy-Based Distributed Systems Management," In *IEEE Transactions on Software Engineering - 25(6)852-869, 1999*.
[16] K. Apt, H. Blair, and A. Walker, "Towards a Theory of Declarative Knowledge," in *Foundations of Deductive Databases*, J. Minker, Ed. MorganKaufmann, 1988, pp.89-148.
[17] G. Jager and R. F. Stark, "The Defining Power of Stratified and Hierarchical Logic Programs," *Journal of Logic Programming*, 15(1 & 2):55-77, 1993.
[18] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and Expressive Power of Logic Programming," 12th IEEE Conf. on Computational Complexity, Ulm, 1997.