Policy Refinement for IP Differentiated Services Quality of Service Management

Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, ***Member, IEEE***
Paris Flegkas, Marinos Charalambides, George Pavlou, ***Member, IEEE***

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. The approach presented here can be used in other application domains such as storage area networks or security management.

Index Terms– Policy refinement, Goal refinement, Refinement patterns

I. 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. In such a network a packet might be handled differently at each hop based on the DiffServ class to which it belongs. 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. Many policy languages have been proposed, but techniques for refining high level goal into implementable policies, amenable to analysis for consistency, 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 Service Level Specifications (SLSs) which have to be satisfied by the network, as well as the derived QoS policies required to satisfy the SLSs, will change frequently. This process of deriving policies from the SLSs is recognized as one of the most difficult research challenges and is not fully automatable; however, techniques and tool support for refinement can be developed. 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 generic automated refinement is not achievable, useful, semi-automated tools can be achieved by constraining the problem to a well defined functional area, such as QoS management, where application specific knowledge can be encoded and used.

This paper extends our previous work on policy refinement for DiffServ QoS management [2] to include techniques for reusing results of the refinement approach. In particular, we look at the use of application specific policy refinement patterns. Additionally, we present details of the tools that have been developed to support the refinement technique. Whilst the refinement technique is generally applicable, this paper focuses on our efforts on identifying the goals, strategies and policies relating to DiffServ QoS management using the TEQUILA framework [3]. TEQUILA uses DiffServ, with Multi-Protocol Labelled Switching (MPLS) [4] to provide dynamic adaptation to varying traffic requirements.

Policy refinement is the process of transforming a high-level, abstract policy specification into a low-level, concrete one. Moffett and Sloman [5], identify the main objectives of a policy refinement process as:

(1) Determine the resources needed to satisfy the policy.
(2) Translate high-level policies into operational policies.
(3) Verify that lower level policies meet the high-level policy requirements.

Objective (1) involves mapping abstract entities defined as part of a high-level policy to concrete objects/devices that make up the underlying system. (2) specifies the need to ensure that derived operational policies are in terms of operations supported by the underlying system. (3) requires a process for incrementally decomposing abstract requirements into more concrete ones, ensuring that at each stage the decomposition is correct and consistent. We propose 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 [6]. These strategies can then be used in specifying policies that can be enforced by

Manuscript received July 17, 2005. Work supported by EPSRC (Grant Nos: GR/R31409/01 and GR/S79985/01), and IBM Research

Arosha K. Bandara is with the Department of Computing, Imperial College London, London, SW7 2AZ, UK (phone: +44-207-594-8235; fax: +44-207-581-1024; e-mail: a.k.bandara@imperial.ac.uk).

E. C. Lupu, A. Russo, N. Dulay, and M. Sloman are all with the Department of Computing, Imperial College London, London, SW7 2AZ, UK (e-mail: {e.c.lupu, a.russo, n.dulay, m.sloman}@imperial.ac.uk).

P. Flegkas, M. Charalambides, and G. Pavlou are all with the Centre for Communications Systems Research, University of Surrey, Guildford, GU2 7XH, UK (e-mail: {p.flegkas, m.charalambides, g.pavlou}@eim.surrey.ac.uk).

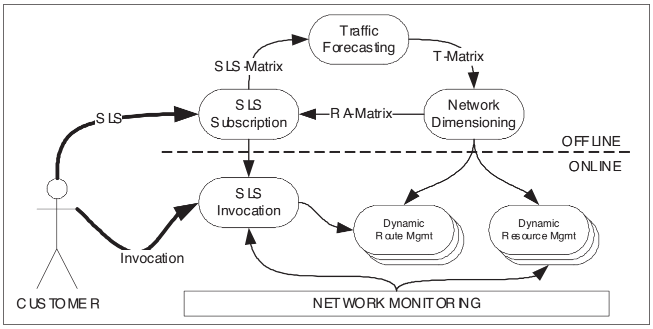


FIGURE 1. *The TEQUILA DiffServ QoS management framework operates in an offline mode to determine the network configuration required to meet long term traffic needs; and an online mode that adapts to short-term variations.*

the system to achieve the original goal. The goal elaboration technique makes use of the KAOS [7] requirements engineering approach, which is based on the use of formal specifications in conjunction with goal elaboration patterns that are proven to be correct. Goal elaboration patterns can 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 presents an overview of our policy refinement technique together with examples of its application. It focuses on the underlying formalisms used in the refinement technique in order to show the derivation and soundness of the procedure. However, the tools we have developed to support the refinement procedure mean that users need only have knowledge of the application domain. To this end, we present ways of reusing results of the policy refinement procedure and go on to discuss its limitations, complexity and applicability.

In the next section we present background information about the TEQUILA framework and our policy refinement approach followed by details of the implementation of the policy analysis and refinement tool in section 3. Section 4 presents some example scenarios of policy refinement in the QoS management domain and in section 5 we describe the use of application-specific patterns for reusing refinement results and automating the refinement procedure. Section 6 presents related work followed by a discussion in section 7. Finally section 8 presents our conclusions and future work.

II. BACKGROUND

A. 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 subscription, Traffic Engineering and Monitoring. SLS subscription 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 1 shows a logical representation of this architecture. The TEQUILA framework has been previously presented [8, 9], so we describe here only the behaviour of the Service Level Specification subscription (SLS-S) and Dynamic Resource Management (DRsM) components which are used in the scenarios presented in the next section.

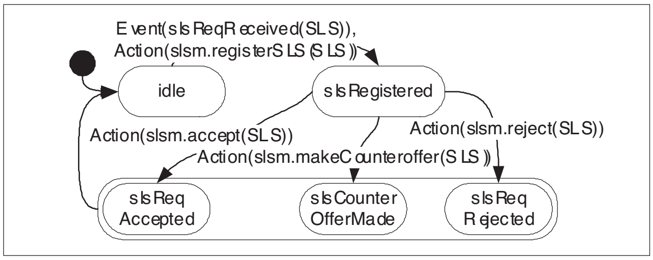


FIGURE 2. *The Service Level Specification Subscription (SLS-S) module is responsible for handling new SLA subscription requests, deciding to accept, reject or make counteroffers based on the policies of the QoS management system.*

The SLS-S module performs admission control, calculates counter-offers and updates traffic forecasts using 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 of overwhelming the network with traffic such that QoS cannot be guaranteed is too high. A state chart model of this behaviour is shown in Figure 2. A more detailed description of the subscription admission control algorithm can also be found in [9].

The DRsM module is a distributed component responsible for reconfiguring the routers in response to short term variations in traffic. It is triggered by network monitors that track PHB utilization and raise threshold-crossing alarms when the bandwidth consumed by a PHB exceeds an upper threshold or drops below a lower threshold. 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 configures the link appropriately; or triggers a new resource provisioning cycle if sufficient bandwidth cannot be allocated. Policies determine how to calculate the new values, configure the link or trigger a new RPC.

B. 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 subgoals, 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 determining if the goal can be achieved by the system. If a particular goal cannot be achieved, then we have to either increase the system's functionality by adding additional management procedures and services, or manually decompose the goal into appropriate lower-level goals. In most situations we would expect the user to do the latter.

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

TABLE 1. *Application-independent goal elaboration patterns.*

KAOS [7] is a technique for goal elaboration, where each goal is represented as a Temporal Logic rule and elaboration patterns are used to decompose the original goal into a set of sub-goals. High-level representations of the goals can be used to shield the users from the formal specification and reasoning processes that are used in the background. Whilst KAOS does not provide automated support for goal elaboration, it does define a library of application-independent elaboration patterns that have been logically proved correct. Table I 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).

In our implementation, these patterns are encoded in the underlying formalism such that when a user provides a high-level goal, the system can infer the sub-goals that are valid decompositions.

For example, given the elaboration patterns presented in Table I, if the user presents the goal “on receiving a SLS from AOL, the SLS should be accepted,” the system would suggest the following sub-goal decompositions:

```
GP1:  on (receive SLS from AOL) then eventually (?Goal1?) AND
      ?Goal1? then eventually (SLS accepted)
GP2:  Goal1? then eventually (SLS accepted) AND
      ?Goal2? then eventually (SLS accepted) AND
      on (receive SLS from AOL) then (?Goal1? OR ?Goal2?)
GP2’: (on (receive SLS from AOL) then eventually (?Goal1?) AND
      ?Goal1? then eventually (SLS accepted) )
   OR (on (receive SLS from AOL) then eventually (?Goal2?) AND
      ?Goal2? then eventually (SLS accepted) )
```

The decompositions also have high-level descriptions of the patterns applied. The user then uses his domain knowledge to choose the decomposition for which the missing goals (denoted by ?GoalX?) can be specified in a meaningful way.

For a particular application domain, specialised goal elaboration patterns can be defined by an expert such that the high-level goals and associated decompositions are expressed using application specific terms. Additionally, the expert can specify invalid sub-goal combinations by using integrity constraints. Both of these techniques limit the set of derived decompositions to those that are applicable and valid for the application domain. This makes it easier for the user to select a suitable decomposition without needing to understand the underlying formalisms.

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 operations that must be performed by the system to achieve a particular 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: $SD, S \vdash G$

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. Guards are specified for transitions with pre-conditions for invoking the operation. We have chosen Statecharts for two reasons: first, because it is unlikely that system descriptions will be provided 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. It is important to note that the system description information need not be provided by the user. Instead, the statechart description of the system may be part of a standard information model or may be provided by equipment vendors.

Given the rules describing a system (SD) and the definition of some desired system state (i.e., the goal — G), abductive reasoning 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 the abductive reasoning process 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.

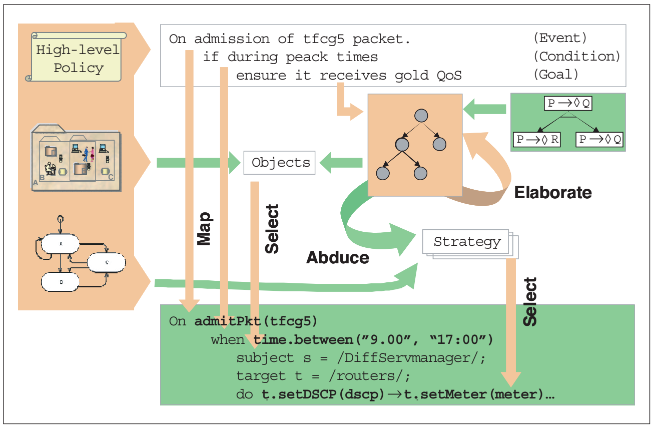
(2) If the system supports multiple strategies for achieving a given goal, each of these strategies could be encoded in a separate policy. This situation might arise when the abductive process yields multiple solutions.

(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 [10]. This domain hierarchy can be derived using automated discovery techniques, a capability of commercial tools such as HP OpenView, CA UniCentre and IBM Tivoli.

Additionally, it is possible to use authorisation policy information and object type information to identify the concrete objects/devices to be specified in the low-level policy. By combining this approach to identify the concrete objects with the goal elaboration and strategy derivation techniques, the overall policy refinement process can be summarized as follows.

The user provides the high-level policy they are interested



```
On admission of tfcg5 packet.                    (Event)
    if during peack times                        (Condition)
        ensure it receives gold QoS              (Goal)
```

```
On admitPkt(tfcg5)
    when time.between("9.00", "17:00")
        subject s = /DiffServmanager/;
        target t = /routers/;
        do t.setDSCP(dscp)→t.setMeter(meter)...
```

FIGURE 3. *The Service Level Specification Subscription (SLS-S) module is responsible for handling new SLA subscription requests, deciding to accept, reject or make counteroffers based on the policies of the QoS management system.*

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

Once a strategy is identified, it is used in the action clause of the final policy. The domain hierarchy is used to identify the subject and target objects in the system for the derived policy that correspond to those entities mentioned in the high-level policy. Finally the event and constraints of the high-level policy are mapped, by the user, into the final policy which is written in a notation that does not require knowledge of the formalisms used (Figure 3). 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 since 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. In the next section we present a formal representation of the managed system and this is followed by a description of the tool that has been developed to support the refinement process.

C. Formal Representation of Managed Systems

As described above, the refinement approach we have developed uses a formal representation of the managed system together with abductive reasoning to automatically derive the strategies for achieving a given goal. In this section we describe the formal language we use to represent the managed system.

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. This predicate is useful when defining static rules (e.g. state constraints)
initiallyTrue(B)	fluent B is initially true.
initiallyFalse(B)	fluent B is initially false.

FIGURE 4. *Event Calculus base predicates*

We use Event Calculus (EC) [11] as the underlying formalism since it has well understood semantics; supports all modes of logical reasoning, including abduction; and the information we are interested in modelling involves events and temporal relationships. Because EC supports a representation of time that is independent of any events that might occur in the system, it is a particularly useful way to specify a variety of event-driven systems. Since its initial presentation, a number of variations of the Event Calculus have been presented in the literature [12]. In this work we use the form presented in [13], 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, which are used to define some auxiliary predicates; and domain independent axioms. These are summarised in Figure 4.

This is the classical form of the Event Calculus where theories are written using Horn clauses. We use pos and neg functions on the fluents to allow us to keep open the interpretation of fluents being true/false. When implementing the formalism using Prolog, circumscription is used to complete predicates, except for holdsAt, initiallyTrue and initiallyFalse. This approach allows the representation of partial domain knowledge (e.g. the initial state of the system). The correspondence between the classical EC with circumscription and the logic program implementation can be found in [12].

The Event Calculus supports deductive, inductive and abductive reasoning. Deduction uses the description of the system behaviour together with the history of events occurring in the system to derive the fluents that will hold at a particular point in time. Induction derives the descriptions of the system behaviour from a given event history and information about the fluents that hold at different points of time. However, the reasoning technique that is of particular interest to our work is abduction. Given the descriptions of the behaviour of the system, abduction can be used to determine the sequence of events that need to occur such that a given set of fluents will hold at a specified point in time.

UML is used as a high-level notation for describing the managed systems because it is widely adopted standard and is supported by many commercial tools. The rest of this section outlines how the UML statechart description of a managed object can be translated into Event Calculus.

The formal language used is based on that described in [14], where in addition to the base predicates and axioms of Event Calculus we make use of the function symbols shown in Table II. The dynamic model of a managed object describes its run-time behaviour in terms of the changes in state caused by performing the management operations. It is important to note that the formalisation of the managed object behaviour need only include the stage changes relating to the objects'

Symbol	Description
`mgdObj(Obj, ClassName)`	Predicate used to indicate that `Obj` is an instance of the managed object type denoted by ClassName.
`pot_state(Obj, Attr, Value)`	Function used to represent the potential value of an attribute of an object in the system. It can be used in an `initiallyTrue` predicate to specify the initial state of the system and also as part of rules that define the effect of actions.
`op(Obj, OpName, Parms)`	Function used to denote the operations specified in an action event (see below)
`systemEvent(Event)`	Function that represents any event that is generated by the system at run-time. The Event argument specified in this term can be any application specific function symbol.
`doAction(ObjSubj, op(ObjTarg, OpName, Parms))`	Function that represents the event of the action specified in the operation term being performed by the subject, `ObjSubj`, on the target object, `ObjTarg`.

TABLE 2. *Predicates and function symbols for formal representation of managed systems.*

management interface. As mentioned previously, this information can be provided as part of a standard information model or equipment vendors.

In order to model the behaviour of the management operations, we specify their pre- and post-conditions. Performing an operation on the system will modify the state of the system in such a way that, once the operation is complete, there will be some new fluents that hold, and some other fluents that cease to hold. This is represented using the initiates and terminates predicates, which are defined in the Event Calculus, according to the following schema:

```
initiates(doAction(_, op(ObjTarg, Action, Parms)),
          PostTrue, Tm) ←
          PreCondition ∧ mgdObj(ObjTarg, ClassName).
terminates(doAction(_, op(ObjTarg, Action, Parms)),
          PostFalse, Tm) ←
          PreCondition ∧ mgdObj(ObjTarg, ClassName).
```

The first rule above states that when the doAction event occurs at time, `Tm`, if the `PreConditions` are true, then the fluent defined by `PostTrue` will hold after that time. Under the same conditions, the second rule states that the fluent defined by `PostFalse` will cease to hold after time, Tm.

Typically, the latter rule is used to invalidate the old value of an object attribute when it changes as a result of the system moving to a new state. In both of these rules, the PreCondition will be represented by a conjunction of holdsAt predicates. The `mgdObj(ObjTarg, ClassName)` predicate in the body of each rule indicates that the rule defines an operation for the type, ClassName.

It is possible to transform this state chart into the Event Calculus notation presented previously where the input shown on each transition arrow is the operation being performed; for transitions between different states, the any attributes that change become the PostFalse fluents; next state values become the PostTrue fluents; and the current state values, together with any guard expressions, become the PreConditions. In order to avoid problems with recursion and infinite looping, self-transitions are omitted from the formal representation.

So following this scheme, Figure 5 shows the formal representation derived from the state chart representation of the SLS-S module shown in Figure 2. Here Rule 1 defines the initial state of the SLSModule object to be `'init'`. The next rule specifies that the occurrence of the `reqReceived(sls)` event and the `registerSLS(sls)` event cause the state to become `'slsRegistered'`. This transition requires that the object ceases to be in the `'init'` state and this is specified by the terminates(…) predicate in Rule 3. The remaining transitions of the statechart are defined in the similar manner in Rules 4–9. These rules also show how we make use of the `pos()` function described previously.

```
1- initiallyTrue(pot_state(Obj, status, 'init'))syms ←
      mgdObj(Obj, 'classSLSModule').
2- initiates(doAction(_, op(Obj, registerSLS, parms(sls))),
      pot_state('d_mgdObjsd_slsmgr', status, 'slsRegistered'), T)←
         holdsAt(pos(pot_state(Obj, status, 'init')), T),
         happens(sysEvent(reqReceived(sls)), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

3- terminates(doAction(_, op(Obj, registerSLS, parms(sls))),
      pot_state(Obj, status, 'init'), T) :-
         holdsAt(pos(pot_state(Obj, status, 'init')), T),
         happens(sysEvent(reqReceived(sls)), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

4- initiates(doAction(_, op(Obj, makeCounteroffer, parms(sls))),
pot_state(Obj, status, 'slsCounterofferMade'), T) ←
      holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

5- terminates(doAction(_, op(Obj, makeCounteroffer, parms(sls))),
      pot_state(Obj, status, 'slsRegistered'), T) ←
         holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

6- initiates(doAction(_, op(Obj, reject, parms(sls))),
      pot_state(Obj, status, 'slsReqRejected'), T) ←
         holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

7- terminates(doAction(_, op(Obj, reject, parms(sls))),
      pot_state(Obj, status, 'slsRegistered'), T) ←
         holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

8- initiates(doAction(_, op(Obj, accept, parms(sls))),
      pot_state(Obj, status, 'slsReqAccepted'), T) ←
         holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).

9- terminates(doAction(_, op(Obj, accept, parms(sls))),
      pot_state(Obj, status, 'slsRegistered'), T) ←
         holdsAt(pos(pot_state(Obj, status, 'slsRegistered')), T),
         mgdObj(Obj, 'classSLSModule'),
         time(T).
```

FIGURE 5. *Formal representation of SLS-S Module behavior.*

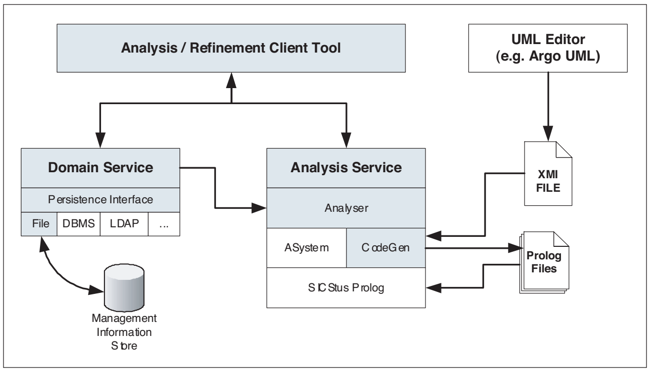


FIGURE 6. *Formal representation of SLS-S Module behavior.*

III. POLICY ANALYSIS AND REFINEMENT TOOL

The technique for policy refinement presented here is based on formal methods, which by their very nature can be difficult to use. The formal specification of the system and policies can be particularly verbose and the results generated from the strategy derivation process are not easy to interpret. Therefore it is important that adequate tool support is provided for administrators and that the tool developed helps them to easily specify the organization and behaviour of the managed system, together with the goals and policies that apply.

A key requirement of the tool is to make the underlying formal notation and reasoning techniques usable. To this end, we allow all information regarding managed objects and their behaviours to be specified in UML using any available editor (e.g. Rational Rose, ArgoUML etc). Then, by using the standard XML Meta-data Interchange (XMI) format of UML, it is possible import these specifications into the policy analysis and refinement tool. This tool can handle the specification of policies, goals, domain hierarchy and the generation of analysis results.

Figure 6 shows the overall architecture of the tool for policy analysis and refinement. As can be seen, there are 3 principal components — the domain service; the analysis service; and the user-interface client application.

A. Domain Service

The domain service provides functionality for storing and retrieving information that describes the entire managed system. In addition to the domain hierarchy itself, this includes the policies, managed objects and goals. Figure 7 shows a detailed class diagram of the objects stored in the domain service, together with the programming interface provided to the other components for accessing the domain hierarchy. The base class for all types of information stored in the domain service is called `ManagedEntity` and this class has `entityID`, `name`, `domainPath` and description attributes which are inherited by all the subclasses. For a given managed entity instance, the entityID attribute is assumed to be a unique identifier.

Additionally, results from analysis activities are stored in objects of type AnalysisResult and associated with the managed entity to which they apply. As shown, a managed entity can be a domain, policy, managed object or goal.

B. Analysis Service

The analysis service deals with the requirements of translating

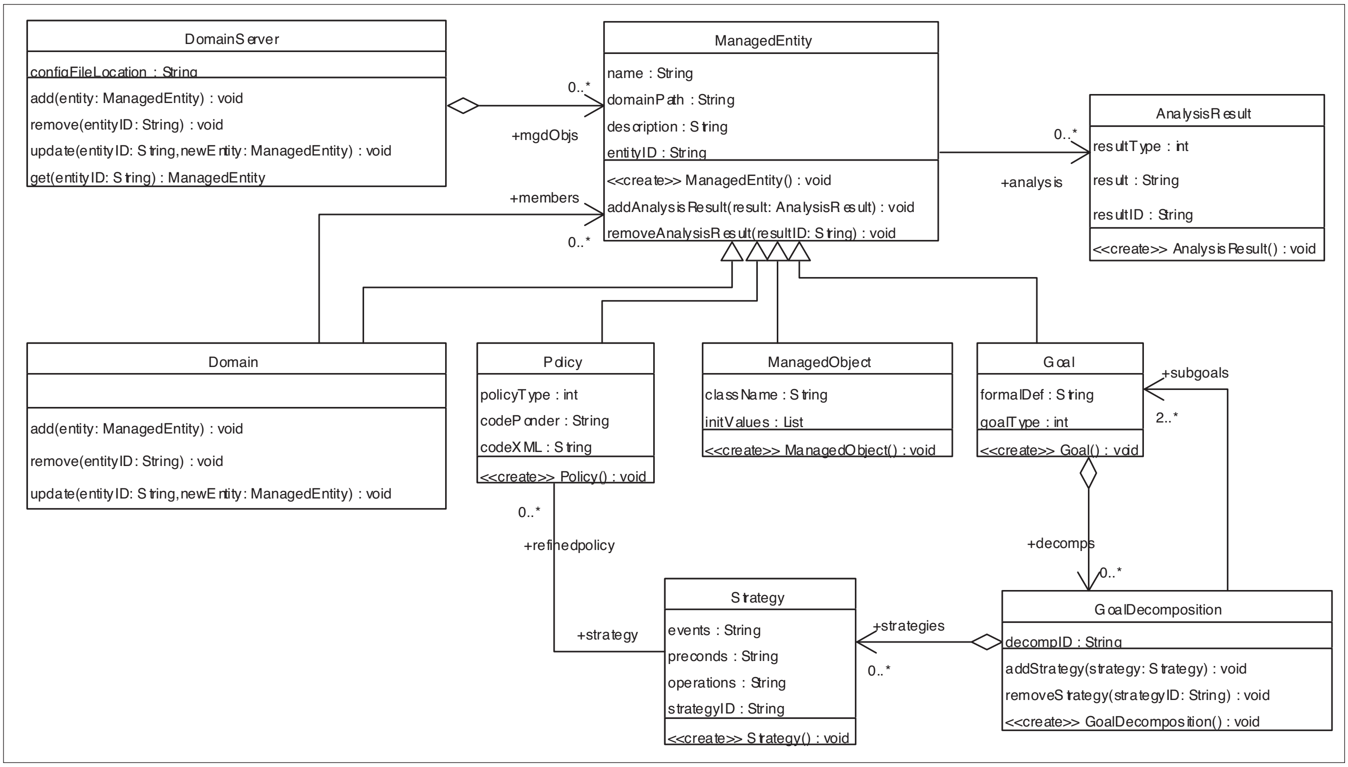
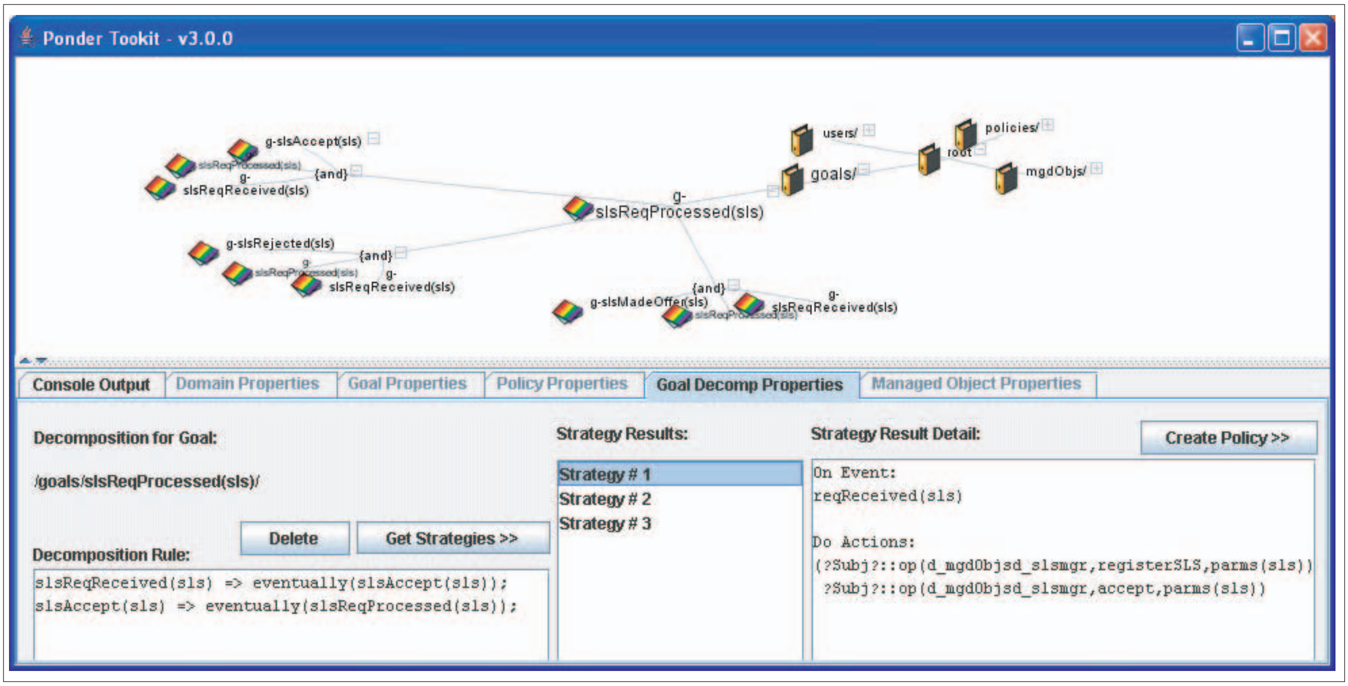

DomainServer
configFileLocation : String
add(entity: ManagedEntity) : void
remove(entityID: String) : void
update(entityID: String,newEntity: ManagedEntity) : void
get(entityID: String) : ManagedEntity
0..*
+mgdObjs
+members
0..*
ManagedEntity
name : String
domainPath : String
description : String
entityID : String
<<create>> ManagedEntity() : void
addAnalysisResult(result: AnalysisResult) : void
removeAnalysisResult(resultID: String) : void
0..*
+analysis
AnalysisResult
resultType : int
result : String
resultID : String
<<create>> AnalysisResult() : void
Domain
add(entity: ManagedEntity) : void
remove(entityID: String) : void
update(entityID: String,newEntity: ManagedEntity) : void
Policy
policyType : int
codePonder : String
codeXML : String
<<create>> Policy() : void
ManagedObject
className : String
initValues : List
<<create>> ManagedObject() : void
Goal
formalDef : String
goalType : int
<<create>> Goal() : void
+subgoals
2..*
0..*
+refinedpolicy
+strategy
+decomps
0..*
Strategy
events : String
preconds : String
operations : String
strategyID : String
<<create>> Strategy() : void
+strategies
0..*
GoalDecomposition
decompID : String
addStrategy(strategy: Strategy) : void
removeStrategy(strategyID: String) : void
<<create>> GoalDecomposition() : void

FIGURE 7. *Classes used by domain service to store managed system information. Domain class is used to organize managed objects and policies; Policy class stores, Ponder code and compiled XML representation of policies in the system; ManagedObject is used to represent devices and managed objects with types designated by the className attribute; Goal has associated GoalDecompositions which in turn have sub-goals; Strategy store management operations, events and pre-conditions associated with a given decomposition. Finally, all managed entities have AnalysisResult objects that store inconsistency information.*



Decomposition for Goal:

/goals/slsReqProcessed(sls)/

Decomposition Rule:

```
slsReqReceived(sls) => eventually(slsAccept(sls));
slsAccept(sls) => eventually(slsReqProcessed(sls));
```

Strategy Results:

Strategy # 1
Strategy # 2
Strategy # 3

Strategy Result Detail:

```
On Event:
reqReceived(sls)

Do Actions:
(?Subj?::op(d_mgdObjsd_slsmgr,registerSLS,parms(sls))
 ?Subj?::op(d_mgdObjsd_slsmgr,accept,parms(sls))
```

FIGURE 8. *Policy refinement tool user interface. The top pane displays the domain hierarchy and the bottom pane has a number of detail tabs. Here we show the detail of a goal decomposition, together with the strategies derived. The decomposition rule (bottom left) describes the relationship between the sub-goals.*

the high-level representations of the policy-based management system into the underlying formalism and generating the analysis and refinement results. To do this the analysis service is integrated with the SICStus Prolog system which provides deductive reasoning capabilities. Abductive reasoning is provided by the A-System abductive proof engine [15] which runs within the SICStus Prolog environment.

Translation of the high-level representations of the domain hierarchy, managed object behaviours, policies and goals is achieved through the use of XML style sheet transformations (XSLT). The `AnalysisServer` class is implemented as a Java RMI server providing methods for analysis tasks like retrieving policy conflict data and performing review queries. Whenever a change is made to information stored in the domain service, the analysis server uses the Analyser class to initiate the process of generating the required Prolog code. The actual translation functions are implemented in the CodeGen component (Figure 6). The `AnalysisServer` is also used to generate the decompositions and strategies for high-level goals.

C. Analysis and Refinement Client

The final component of the analysis and refinement tool is the client application that implements the user interface (Figure 8). The primary view of this application is based on an new version of the Ponder hyperbolic domain browser [10]. It uses the HyperGraph library [16] to implement a hyperbolic viewer to browse a hierarchical domain structure more effectively than a simple tree viewer . It also supports the option of selectively collapsing certain sub-domain hierarchies.

In order to view the detailed information regarding any particular entity in the domain hierarchy, a context sensitive properties pane was used. Whenever the user selects an element of the domain hierarchy, the tabbed panel in the bottom half of the screen shows the details relating to that particular element. For example, Figure 8 illustrates the goal decomposition detail panel, showing the sub-goal information, together with the derived strategies. At present, the tool is a research prototype that aims to demonstrate the practical applications of our approach to policy analysis and refinement. For this reason, some of the specifications shown in the user interface are still presented in the formal notation. However, we expect usage to be considerably simplified in production versions of the tool.

In the next section we present two scenarios where the policy refinement approach and tool described are used to derive a policy to meet the requirements of the TEQUILA framework.

IV. DiffServ Goals, Strategies and Policies

A. Example 1: Admission Control

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 10Mb/s throughput guarantee. SLS[customer: aol; scope: pipe(r1,r6); qos: qosClass(EF, 20, 0); bwReq: bw(10Mb/s)], 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

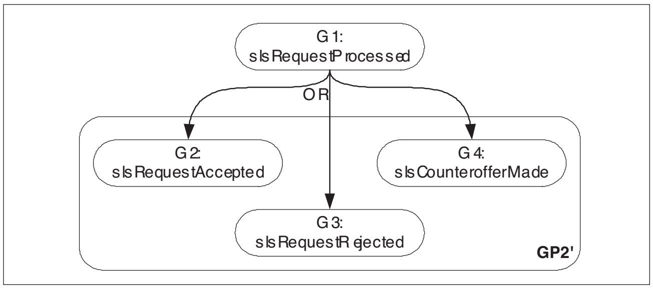


FIGURE 9. *Goal decomposition for SLS subscription scenario.*

the domain-independent pattern GP2' (see Table I) 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).
```

As shown in Figure 9, 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.

Figure 8 shows how this scenario would appear in the policy refinement toolkit. Here the top panel illustrates the goal elaboration hierarchy within the organizational domains of the system. The bottom panel shows the detail of one of the goal decompositions, where the relationship between the sub-goals is shown by the decomposition rule in the bottom left, and the strategies derived are shown to the right.

B. Example 2: Adapting to Traffic Increase

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 between 11am and 1pm and causes a network utilisation greater than 85 percent of the maximum allocation, the bandwidth allocation should be increased by 10 percent 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) should 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 of these goals 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 10 indicates the applicable patterns at each

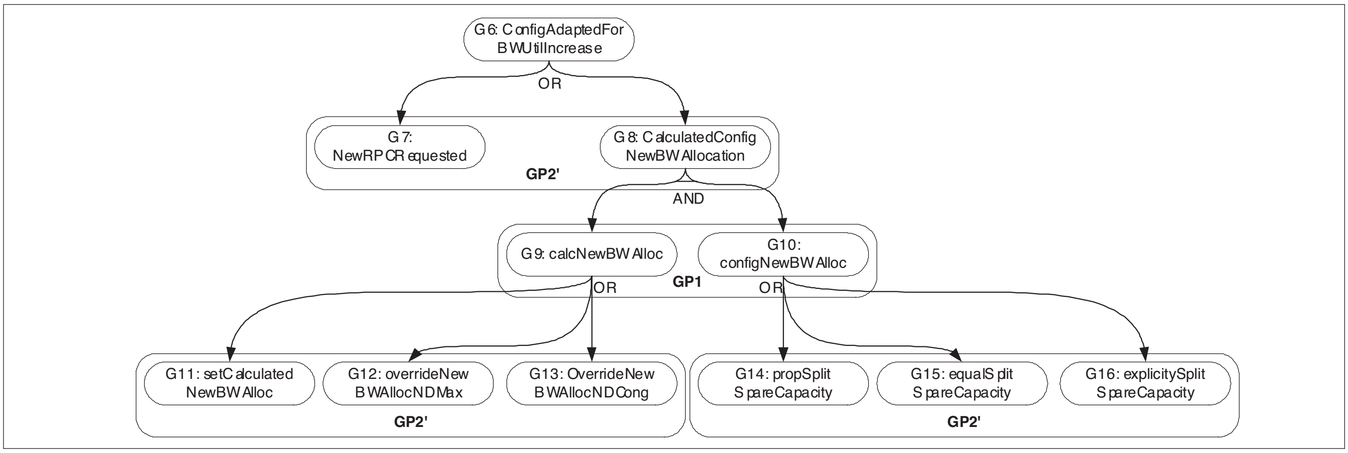

G6: ConfigAdaptedFor BWUtilIncrease
OR
G7: NewRPCRequested
G8: CalculatedConfig NewBWAllocation
GP2'
AND
G9: calcNewBWAlloc
G10: configNewBWAlloc
GP1
OR
OR
G11: setCalculated NewBWAlloc
G12: overrideNew BWAllocNDMax
G13: OverrideNew BWAllocNDCong
GP2'
G14: propSplit SpareCapacity
G15: equalSplit SpareCapacity
G16: explicitySplit SpareCapacity
GP2'

FIGURE 10. *Goal decomposition for traffic increase scenario.*

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 and G15. 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');
    }
```

Note that 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.

V. 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.

Some of this automation results from allowing the reuse of pre-defined goal elaboration patterns when decomposing high-level goals. As described previously, users are presented with a template for the possible low-level goals, together with a high-level description of the pattern being used. This allows users to select suitable decompositions without needing to understand the underlying formal derivation. We extend this idea of reuse by defining refinement patterns that directly relate a goal, to the set of policies that could achieve it. Each pattern is parameterised according to the specifics of the high-level goal. To achieve this, we introduce the following syntax for a *policy refinement pattern* construct:

```
policyPattern patternName(ParameterList) {
   description A description of the policy pattern.
   goalHierarchy goal [refinesTo (goalHierarchy)]
   policies { // Group of policies that will achieve
             // the goals associated with 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:

```
policyPattern /ptn/acceptSLS(String customer, QoSClass qc) {
  description Accept incoming SLS from customer provided it is \
          for a specified QoSClass and bandwidth available.
  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; } } }
```

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.

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 percent), 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 percent));
```

The policy management system instantiates each of the policies in this pattern with the parameters specified, then the overall policy specification can be analysed for inconsistencies as shown in [14].

VI. Related Work

There are few practical studies on policy refinement. Power [17] 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 [18]. 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 can be fully automated, but depends on the correctness of the table which requires domain expertise.

This technique is similar to the case-based reasoning approach to policy transformation proposed by researchers at IBM [19] where table look-ups are used to match high-level requirements parameters to device level configuration values. For example, by building a database of the average response times of a web-server farm containing different numbers of servers, case-based reasoning can be used to determine the number of servers that should be activated to satisfy a given response time requirement. This approach to policy refinement has limited applicability since it can only be used in those cases for which it is feasible to build a database of the requirements and configuration parameters.

VII. Discussion

The current state of the art in systems management requires administrators to be familiar with the intricate details of the equipment they manage and to often perform configurations manually. In enterprise environments where the management tasks span different levels of abstraction from applications and services to physical devices; and are highly heterogeneous, administration becomes increasingly difficult. Policy-based management allows administrators to change the management strategy of a system by changing policies dynamically rather than reimplementing management functionality.

Effective systems management requires the ability to verify properties of the system. In particular it is necessary to analyse policies to detect inconsistencies. After preliminary work on modality and application specific conflicts [20], we have shown how an Event Calculus representation of both policies and managed systems can be used, together with abductive reasoning for policy analysis [14]. 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 the analysis and 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 [21]. Stratified logic programs are decidable in polynomial time [22].

The work presented in this paper has shown how to partially automate the refinement of policies whilst hiding the details of the underlying formal techniques from the user. Achieving this objective whilst also providing some degree of consistency checking and automated reasoning capability requires the use of models. The refinement procedure requires some user intervention, e.g. to map constraints associated with goals into the final policies. However, this only requires users to be familiar with the models of the resources being managed, not the underlying formalisms being used to support the refinement process. This task will become easier when standard information models (e.g. CIM) are adopted.

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, identifies the objects required to execute strategies. These features illustrate how this solution satisfies the objectives of policy refinement identified in [5].

In addition to meeting these functional objectives, a key challenge in developing a policy refinement approach is to achieve an acceptable trade-off between the generality of the approach and the level of automation possible. Fully-automated approaches are also highly specialised to particular applications domains and cannot be applied to other domains. On the other hand, generalised approaches to refinement require experts who are familiar with both the application domain and low-level formal representations to provide information regarding the managed system.

Our approach addresses this challenge by providing a general refinement procedure that can be specialised by libraries of reusable application specific patterns. Also, 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 goal and refinement patterns can be mostly driven by their natural language description. The tools we are developing aim to minimise the amount of required knowledge and intervention from network operators.

Other open challenges related to policy refinement include how to derive additional policy information such as the parameter values to be used with the management operations, low-level events and constraints; how to better guide the user in selecting appropriate goal decompositions and strategies when multiple solutions are derived; and how to optimise the refinement procedure such that it is feasible to use it for runtime decision making in a policy-managed environment. This last item is particularly relevant in the context of autonomic computing, where the aim is to have systems that are capable of self-management.

VIII. 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. Additionally, we have described the tool that has been developed to support this refinement process and shown how it is used to derive the strategies for relating to an admission control scenario of the TEQUILA DiffServ framework. 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.

As mentioned, an area of policy refinement that requires further study is the derivation of parameter values for management operations, e.g. the bandwidth to be allocated for a particular link. We are currently working on integrating constraint logic programming techniques into our formal reasoning framework to provide such capabilities. Additionally, we hope to provide further automation of the refinement process by leveraging utility functions to rank the goal decompositions and strategies derived by the abductive inference procedure. Finally we are working to improve the usability of the analysis and refinement tools, with particular emphasis on minimising the use of any formal notations in the specification of goals and related information.

References

[1] M. Carlson *et al.*, "An Architecture for Differentiated Services," Network Working Group — RFC2475, http://www.ietf.org/ rfc/rfc2475.txt, 1998.

[2] A. K. Bandara *et al.*, "Policy Refinement for DiffServ Quality of Service Management," in *Proc. 9th IFIP/IEEE Int'l. Symp. Integrated Network Management*, Nice, France, 2005.

[3] P. Flegkas, P. Trimintzios, and G. Pavlou, "A Policy-based Quality of Service Management Architecture for IP DiffServ Networks," *IEEE Network Mag.*, special issue on *Policy Based Networking*, vol. 16, no. 2, pp. 50–56, 2002.

[4] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," Network Working Group — RFC3031, http://www.ietf.org/rfc/rfc3031.txt, 2001.

[5] J. Moffett and M. S. Sloman, "Policy Hierarchies for Distributed Systems Management," *IEEE Journal Select. Areas in Commun.*, vol. 11, no. 9, special issue on *Network Management*, pp. 1404–1414, 1993.

[6] A. K. Bandara, E. C. Lupu, and A. Russo, "A Goal-based Approach to Policy Refinement," in *Proc. 5th IEEE Int'l. Workshop Policies for Distributed Systems and Networks*, IBM TJ Watson Research Centre, New York, USA, IEEE Press, June 2004.

[7] R. Darimont and A. van Lamsweerde (1996). "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," *4th ACM Symp. Foundations of Software Engineering (FSE4)*, pp. 179–190, 1996.

[8] P. Trimintzios *et al.*, "Service-driven Traffic Engineering for Intra-domain Quality of Service Management," *IEEE Network Mag.*, special issue on *Network Management of Multiservice, Multimedia, IP-based Networks*, vol. 17, no. 3, pp. 29–36, 2003.

[9] E. Mykoniati *et al.*, "Admission Control for Providing QoS in IP DiffServ Networks: the TEQUILA Approach," *IEEE Commun. Mag.*, vol. 41, no. 1, 2003.

[10] N. Damianou *et al.*, "Tools for Domain-based Policy Management of Distributed Systems," in *Proc. 8th IEEE/IFIP Network Operations and Management Symp.*, Florence, Italy, Apr. 2002.

[11] R. A. Kowalski and M. J. Sergot, "A Logic-Based Calculus of Events," *New Generation Computing*, vol. 4, pp. 67–95, 1986.

[12] R. Miller and M. Shanahan, "The Event Calculus in Classical Logic — Alternative Axiomatisations," *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, A. Kakas and F. Sadri, Springer, vol. 2048, pp. 452–490, 1999.

[13] A. Russo *et al.*, "An Abductive Approach for Analysing Event-Based Requirements Specifications," in *Proc. 18th Int'l. Conf. Logic Programming (ICLP)*, Copenhagen, Denmark, July 2002.

[14] A. K. Bandara, E. C. Lupu, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," in *Proc. 4th IEEE Int'l. Wksp. Policies for Networks and Distributed Systems (Policy 2003)*, Lake Como, Italy, IEEE, June 2003.

[15] B. van Nuffelen and A. Kakas, "A-System : Programming with Abduction," in *Proc. Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, Springer-Verlag, 2001.

[16] Hypergraph, "Hypergraph API," http://hypergraph.sourceforge.net

[17] M. Casassa Mont, A. Baldwin, and C. Goh, "POWER Prototype: Towards Integrated Policy-Based Management," Technical Report: HP Laboratories Bristol, Bristol, UK, 1999.

[18] D. C. Verma, Policy-Based Networking: Architecture and Algorithms, New Riders Publishing, 2001.

[19] M. S. Beigi, S. Calo, and D. Verma, "Policy Transformation Technqiues in Policy-based Systems Management," in *Proc. Int'l. Wksp. Policies for Distributed Systems and Networks*, Yorktown Heights, New York, IEEE, June 2004.

[20] E. C. Lupu and M. S. Sloman, "Conflicts in Policy-Based Distributed Systems Management," in *IEEE Trans. Software Engineering*, special issue on *Inconsistency Management*, vol. 25, no. 6, pp. 852–869, 1999.

[21] K. R. Apt, H. A. Blair, and A. Walker, "Towards a Theory of Declarative Knowledge," *Foundations of Deductive Databases*, J. Minker, San Mateo, CA, Morgan Kaufmann, pp. 89–148, 1988.

[22] E. Dantsin *et al.*, "Complexity and Expressive Power of Logic Programming," in *Proc. 12th Annual IEEE Conf. Computational Complexity (CCC'97)*, Ulm, Germany, IEEE Press, June 1997.

BIOGRAPHIES

Dr. Arosha Bandara completed his PhD in Policy-based Systems Management at the Department of Computing at Imperial College London in September 2005. He continues to work there as a post-doctoral researcher, working on policy analysis for QoS management. He received a MEng in Information Systems Engineering from Imperial College London in 1998 and worked for Sapient Corporation until 2001. His research interests include the application of formal analysis techniques to distributed systems management, autonomic computing and network security.

Dr. Emil Lupu is a Senior Lecturer in the Department of Computing at Imperial College London where he leads several research projects in the areas of policy-based network and systems management, pervasive computing, trust and security. He has over 50 publications in these areas and serves on the steering committee of the IEEE Workshop on Policies for Distributed Systems and Networks and on the program committee of several international conferences. See http://www.doc.ic.ac.uk/~ecl1 for further details and selected publications.

Dr. Alessandra Russo is a Lecturer at the Department of Computing, Imperial College London. Her research expertise is across the areas of Artificial Intelligence and Software Engineering. She has over 40 publications in international conferences and journals, one of which was awarded prize for best application paper. She serves on the PC of various international conferences, chaired the (ASE) Workshops on Intelligent Technologies for Software Engineering (2004), and been co-chair of the BCS Requirements Engineering Specialist Group.

Professor Morris Sloman is Director of Research and Deputy Head of the Department of Computing, Imperial College London. His research interests include management of networks and distributed systems, adaptive security management, trust and security for pervasive systems, autonomic management for pervasive systems. He is on the steering committees for the conferences on Policies for Distributed Systems and Networks, Integrated Management (IM), Network Operations and Management (NOMS). See http://www.doc.ic.ac.uk/~mss for more details and selected papers.

Dr. Naranker Dulay is a Senior Lecturer in the Department of Computing, Imperial College London. He obtained his BSc in Computer Science from the University of Manchester and his PhD from Imperial College. His current research interests lie the areas of languages and architectures for distributed, mobile and pervasive systems, particularly to address security, trust and context-awareness.

Dr. Paris Flegkas completed his PhD in the Networks Research Group at the Center for Communication Systems Research, University of Surrey in July 2005. He received a Diploma in electrical and computer engineering from Aristotle University, Thessaloniki, Greece, and a M.Sc. in Telematics (communications and software) from the University of Surrey. His research interests are in the areas of policy-based management, network and service management technologies, and IP QoS provisioning.

Marinos Charalambides received a B.Eng. (First Class Hons.) in Electronic and Electrical Engineering and an M.Sc. (Distinction) in Communications Networks and Software, both from the University of Surrey, UK, in 2001 and 2002 respectively. He is currently a Ph.D. candidate at the Centre for Communication Systems Research, University of Surrey, and his research interests are in the areas of policy-based management, policy analysis, and IP quality of service.

Professor George Pavlou works at the Center for Communication Systems Research, University of Surrey where he leads the activities of the Networks Research Group. He holds a Diploma in Engineering from the National Technical University of Athens, Greece, and M.Sc. and Ph.D. degrees in computer science from University College London. His research interests focus on network management, networking, and service engineering, covering aspects such as traffic engineering, QoS management, policy-based systems, programmable networks, and communications middleware.