Proceedings 5th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2004)
IBM TJ Watson Research Centre, New York, USA, June 2004

A Goal-based Approach to Policy Refinement

Arosha K Bandara[1] Emil C Lupu[1] Jonathan Moffett[2] Alessandra Russo[1]

1: Department of Computing
Imperial College London,
180 Queen's Gate, London SW7 2AZ, UK
{bandara, e.c.lupu, ar3}@doc.ic.ac.uk

2: Department of Computer Science
University of York
Heslington, York YO10 5DD, UK
jdm@cs.york.ac.uk

Abstract

As the interest in using policy-based approaches for systems management grows, it is becoming increasingly important to develop methods for performing analysis and refinement of policy specifications. Although this is an area that researchers have devoted some attention to, none of the proposed solutions address the issue of deriving implementable policies from high-level goals. A key part of the solution to this problem is having the ability to identify the operations, available on the underlying system, which can achieve a given goal.

This paper presents an approach by which a formal representation of a system, based on the Event Calculus, can be used in conjunction with abductive reasoning techniques to derive the sequence of operations that will allow a given system to achieve a desired goal. Additionally it outlines how this technique might be used for providing tool support and partial automation for policy refinement. Building on previous work on using formal techniques for policy analysis, the approach presented here applies a transformation of both policy and system behaviour specifications into a formal notation that is based on Event Calculus. Finally, it shows how the overall process could be used in conjunction with UML modelling and illustrates this by means of an example.

1. Introduction

Policy based approaches to network and systems management are of particular importance because they allow the separation of the rules that govern the behaviour of a system from the functionality provided by that system. This means that it is possible to adapt the behaviour of a system without the need to recode functionality, and changes can be applied without stopping the system. Research into policy based systems management has focussed on languages for specifying policies and architectures for managing and deploying policies in distributed environments. However, whilst there have been some promising developments in the area of policy analysis, policy refinement remains a much-neglected research problem.

Policy refinement is the process of transforming a high-level, abstract policy specification into a low-level, concrete one. Moffett and Sloman [1], identify the main objectives of a policy refinement process as:

- Determine the resources that are needed to satisfy the requirements of the policy.
- Translate high-level policies into operational policies that the system can enforce.
- Verify that the lower level policies actually meet the requirements specified by the high-level policy.

The first of these objectives involves mapping abstract entities defined as part of a high-level policy to concrete objects/devices that make up the underlying system. The second specifies the need to ensure that any policies derived by the refinement process be in terms of operations that are supported by the underlying system. The final objective requires that there be a process for incrementally decomposing abstract requirements into successively more concrete ones, ensuring that at each stage the decomposition is correct and consistent.

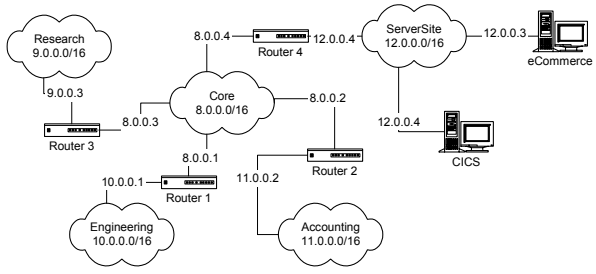

Figure 1: Example enterprise network

Figure 1 presents an example scenario, originally developed by Verma [2], where policy refinement might be applied. Here, an enterprise network must implement a Service Level Agreement (SLA) where one of the

clauses specifies that "WebServices Applications on the eCommerce Server must receive Gold Quality of Service (QoS)". This requirement may be articulated as a policy which states that "On demand, the network should be configured to provide Gold QoS to WebServices applications on the eCommerce Server". Based on the objectives mentioned above, the policy refinement process should transform this high-level policy into lower-level policies that take into account:

1. The specific routers that need to be configured to handle the traffic for "WebServices applications on the eCommerce Server".
2. The set of operations, supported by these routers that will meet the objective of "Gold QoS for WebServices Applications on the eCommerce Server".

And the overall process should meet the third objective of ensuring that there is a means to verify that low-level policies actually meet the requirement defined by the high-level one. This example illustrates that the policy refinement problem is actually composed of two parts:

1. Refinement of abstract entities into concrete objects/devices.
2. Refinement of high-level goals into operations, supported by the concrete objects/devices, that when performed will achieve the high-level goal.

In order to solve these problems we need a formal representation for objects, their behaviour and organisation; a technique for refining high-level goals into more concrete ones; and finally a means of inferring the combination of operations that will achieve these concrete goals. To this end we use the formalism presented in [3] to model the behaviour and organisation of the objects, together with the goal elaboration technique developed by Darimont et al. [4], to refine high-level goals into concrete ones. However, the refined goals cannot be directly used in policies without first identifying the operations that will achieve them. To support this identification process, we introduce the concept of a *strategy*, which is the mechanism by which a given system can achieve a particular goal, i.e., a strategy is the relationship between the system description and the goal. By having a formal specification of the latter two types of information we can use abductive reasoning to infer the strategy.

In keeping with our previous work [3], we propose that the entire formalism be implemented in Event Calculus [5] since this is a particularly suitable notation for modelling the event-driven nature of the systems we are interested in; and also because this allows us to make use of the mapping from the Ponder policy notation to Event Calculus and the conflict detection techniques that we have already developed. We use the goal elaboration technique presented in [4] because it provides the concept of domain-specific and domain-independent refinement patterns, logically proven goal refinement templates that can be easily reused. We can use such patterns to capture the refinement of goals that are commonly encountered in policy-based management, thus simplifying the refinement process for the user.

The paper is organised as follows. Section 2 presents background information on the techniques we are building on to develop our policy refinement solution. Section 3 presents the policy refinement approach together with the details for the formal notation being used; and Section 4 illustrates how the refinement technique might be applied to the example described above. In Section 5 we discuss the solution, its strengths and weaknesses; and in Section 6 we compare this work with existing research in the field. Finally Section 7 presents some conclusions together with directions for future work.

2. Background

2.1 Goal Elaboration

The first component of the policy refinement process to be considered is a technique for refining high-level goals, defined during the requirements gathering process, into concrete low-level policies. Figure 2 shows how the requirements of a system might be refined from high-level goals into implementable classes/modules. The decomposition of a goal can be either conjunctive (i.e. only by achieving all the sub-goals can we consider that the higher-level goal is achieved) or disjunctive (i.e. by achieving any one of the sub-goals we can consider the higher-level goal is achieved).

Note the two distinct phases of the refinement process. The first phase is one of goal refinement where the focus is on translating abstract goals into operationalised goals. An operationalised goal is one that has been assigned to specific agent whose capabilities enable the system to satisfy that goal. These goals are often referred to as the System Requirements. Taking the example presented previously, this process would transform the SLA requirement of providing gold QoS for a particular class of traffic into a set of goals that define the configuration changes that must be applied to the routers in the network. The second phase of the refinement process takes these system requirements and maps them to specific modules/operations that can be implemented within the context of the system architecture. This phase could be considered to be architectural or system design. In our example, this would involve identifying the operations to be invoked on the routers to achieve the desired goal.

In the above scheme, each high-level goal is refined into sub-goals, forming a goal refinement hierarchy where the dependencies between the goals at the different levels of refinement are based on the form of goal decomposition used (AND/OR). Additionally there can be dependencies between goals in different hierarchies. The process of refinement will involve 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 the goal cannot be achieved, it is necessary to elaborate the information at the higher-level such that suitable lower-level goals can be derived.

Work done by Darimont et al. [4], proposes a formal technique for elaborating goals grounded in Temporal Logic. Called KAOS, this approach represents each goal as a Temporal Logic rule and then makes use of refinement patterns to decompose these goals into a set of sub-goals that logically entail the original goal. Additionally, this technique makes use of obstacles (negated goals) which are then elaborated and resolved to provide new goals. This process results in a set of refined goals, and the identification of objects and operations that might operationalise them. The final stage of the procedure is to assign each of the refined goals to a specific object/operation such that the final system will meet the original requirements. Whilst the KAOS approach does not provide any automated support for the goal refinement process, it does define a library of domain-specific and domain-independent refinement patterns that have been logically proved.

A domain-independent goal refinement patterns uses properties of temporal logic operators to provide a proven relationship between a high level goal and a set of sub-goals. For example, the transitivity property of the $\Diamond R$ (R will eventually be true) operator provides the following simple domain-independent goal refinement pattern:

$$(P \Rightarrow \Diamond R,\ R \Rightarrow \Diamond Q) \vdash P \Rightarrow \Diamond Q$$

```
If P is true then eventually R is true, AND
If R is true then eventually Q is true, THEN
If P is true then eventually Q is true.
```

In our example scenario, a domain-specific pattern might be one that describes the sub-goals required to guarantee QoS for a class of application traffic. The user could then refine the goal instance "provide Gold QoS to WebServices applications on the eCommerce Server", by instantiating this pattern with the Gold class of service and the appropriate application type. Once the user has specified appropriate sub-goals based on the particular pattern, the specification is checked for inconsistencies.

Policy-based systems use rules to govern their behavioural choices whilst satisfying the goals of the system. Therefore a policy refinement technique must provide a link between each goal and the underlying system behaviour in order to derive the different ways in which the system can satisfy the goal. This information can then be encoded into policies that control the behaviour of the system as needed. Whilst we can use the KAOS approach to refine abstract goals into lower-level ones, it does not provide a mechanism to connect the goals with the behaviour description of the system. Therefore, in this paper we show how the notation used by KAOS can be combined with state charts, Event Calculus and abductive reasoning to provide a practical refinement technique.

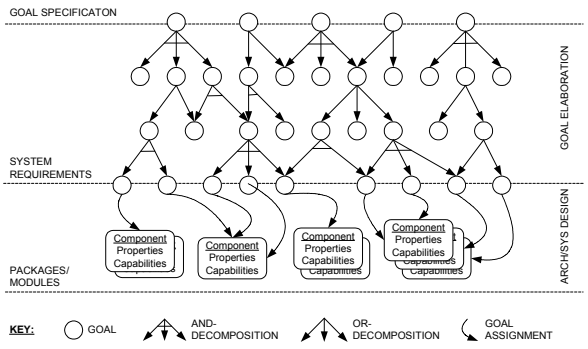

Figure 2: Goal refinement hierarchy

2.2 Event Calculus

We propose to use Event Calculus (EC) 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. Event Calculus is a formal language for representing and reasoning about dynamic systems. Because the language 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 [4], a number of variations of the Event Calculus have been presented in the literature [6]. In this work we use the form presented in [7], 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 3.

This is the classical form of the Event Calculus where theories are written using Horn clauses. The frame problem is solved by circumscription, which allows the completion of the predicates `initiates`, `terminates` and `happens`, leaving open the predicates `holdsAt`, `initiallyTrue` and `initiallyFalse`. This approach allows the representation of partial domain knowledge (e.g. the initial state of the system). Formulae derived by the Event Calculus are in effect classically derived from the circumscription of the EC representation. To provide an implementation of such a calculus in Prolog, we use `pos` and `neg` functors. The semantics of the Prolog implementation assumes the Close World Assumption (CWA) and models are essentially Herbrand models where predicates are appropriately completed. The use of `pos` and `neg` functions on the fluents allows us to keep open the interpretation of fluents being true/false, in the same way as circumscription does in the classical representation. In this way we can guarantee that the implementation of our EC is sound and complete with respect to the classical EC formalisation. The correspondence between the classical EC with circumscription and the logic program implementation can be found in [6].

Base predicates:

initiates(A,B,T)	event A initiates fluent B for all time > T.
terminates(A,B,T)	event A terminates fluent B for all time > T.
happens(A,T)	event A happens at time point T
holdsAt(B,T)	fluent B holds at time point T. 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 3: Event Calculus predicates and axioms

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 aims to derive 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.

3. Policy Refinement Approach

As mentioned previously, the KAOS approach provides a technique for refining abstract goals into lower-level ones. However these low-level goals cannot be directly used in refined policies. To do this, it is necessary to have a method for inferring the mechanism by which the system can achieve a goal at a given abstraction level.

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) - $SD_x, S_x \vdash G_x$

X is a label denoting the abstraction level.

So, in our approach, it is expected that the user would provide 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. The behaviour of the system is defined in terms of the pre- and post-conditions of the operations supported by the components, which the user can specify using a high-level notation such as state charts. Since the goals to be satisfied can be defined in terms of desired system states, they can be specified in a notation similar to that used to specify the post-conditions of the operations.

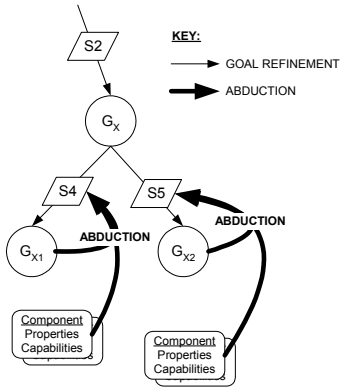

Figure 4: Deriving strategies from goals and system description

Once the user has provided this information, it is first necessary to transform it into a formal representation that supports automated analysis. 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 (Figure 4). Additionally, we can use the properties of the goal decomposition approach described previously to decompose the system description and strategies as follows:

(2) - $G_{X1}, G_{X2}, \ldots, G_{XN} \vdash G_X$ *Goal Decomposition*

(3) - $SD_{X1}, S_{X1} \vdash G_{X1}$
$SD_{X2}, S_{X2} \vdash G_{X2}$ ...
$SD_{XN}, S_{XN} \vdash G_{XN}$ *(from 1)*

This shows that if there is some combination of lower level goals from which we can infer the original goal, then for each of these sub-goals there must be a corresponding strategy and system description combination which will achieve it. Therefore, provided the goal decomposition is correct, intuitively the combination of the lower level system descriptions should allow inference of the abstract system description and similarly the combination of the lower level strategies should allow inference of the abstract strategy.

As mentioned previously, the other component of the refinement process is to refine abstract entities into concrete objects/devices in the system. For example, in the system illustrated in Figure 1, there might be an abstract entity called "Network" that logically consists of the "Engineering Network", "Core Network" etc., where each of these in turn consist of the routers and servers within them. We propose that a domain hierarchy be 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 of a domain is explicit and not defined in terms of a predicate on object attributes. An advantage of specifying policy scope 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. The formal representation of the domain structure is as shown in [3].

In order to implement the approach outlined above, it is necessary to have a formal representation of the system description; and the strategies and goals. However, for the implementation to be usable, it would be ideal to be able to model the systems in a high-level notation and translate this into Event Calculus for analysis purposes. UML would be well suited for this purpose since it is widely used and is supported by many commercial tools.

This rest of this section outlines how UML would represent each of the types of information that need to be modelled together and describes how they can be translated into Event Calculus. The formal language being used is based on that described in [3], where in addition to the base predicates and axioms of Event Calculus we make use of the function symbols shown in Table 1.

3.1 System Description

The system description models the objects in the system in terms of their behaviour. The notation used to formally model the behaviour of objects is identical to that described in [3]. Using this notation, and building on the example used previously, it is possible to illustrate the use of these rules for modelling system behaviour. So, let us say there is an object of type `DiffServRouter` in the example system. This type has attributes to hold the IP interfaces and actions to configure various parameters of the router, which might be represented in UML as a class diagram. The actions for the `DiffServRouter` type can be specified in a UML state chart representation as shown in Figure 5.

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 action being performed. For transition between different states, the current state values become the `PostFalse` fluents; any actions associated with the transition and next state values become the `PostTrue` fluents; and the current state values become the `PreConditions`. Self-transitions should not specify the current state as `PostFalse` fluents. So following this scheme, the transition labelled (**) in Figure 5 would be represented in the Event Calculus as follows:

Table 1: Function symbols.

Symbol	Description
`state(Obj, Vo, Value)`	Represents the value of a variable 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, Action(VP))`	Used to denote the operations specified in an action event (see below)
`systemEvent(Event)`	Represents any event that is generated by the system at runtime. The Event argument specified in this term can be any application specific predicate or function symbol.
`doAction(ObjSubj, op(ObjTarg, Action(VP)))`	Represents the event of the action specified in the operation term being performed by the subject, `ObjSubj`, on the target object, `ObjTarg`.

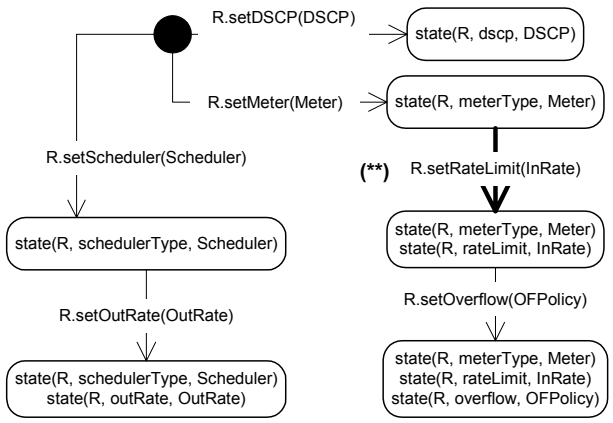

Figure 5: UML state chart for DiffServRouter type

```
initiates(
 doAction(_, op(diffServRouter,
                        setRateLimit(inRate))),
 state(diffServRouter,rateLimit, inRate), T) ←
 holdsAt(pos(state(diffServRouter,
                   meterType, Meter)), T).
```

The above rule also shows how we make use of the `pos()` function as described in Section 2.2. Note that, whilst we have shown the details of the Event Calculus representation of the system organisation and behaviour models, it is not necessary for the user to directly specify anything in the formal notation. Instead they would use UML class diagrams, state charts together with a domain model chart and the system will generate the Event Calculus code required for the refinement procedure.

3.2 Goals

A UML profile for modelling goals and goal refinement patterns described in the KAOS approach has already been developed and is presented in [9]. Figure 6 shows how an AND-decomposition of a goal would be represented in this notation. The profile defines a number of attributes for the <<goal>> stereotype, including one to hold the temporal logic representation of the goal. However, in order to support the formal analysis required for validating the goal refinements, it is still necessary to map the temporal logic formalism of KAOS into Event Calculus and describe a mechanism for verifying the correctness of a goal refinement.

The goal refinement patterns provided by KAOS make use of some of the temporal logic operators described in [10]:

X	X holds in the current state
◇ X,	X will eventually hold
◆ X,	X held at some time in the past
Y ***W*** X,	Y holds ***unless*** X holds

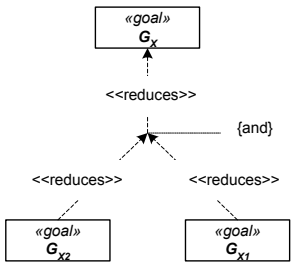

Figure 6: UML representation of the AND-decomposition of a goal

```
   X  -> ∃T:     holdsAt(X, T) ∧ T=now.

◇ X  -> ∃T,T':  holdsAt(X, T') ∧ T=now  ∧
                T'>T.

◆ X  -> ∃T,T':  holdsAt(X, T') ∧ T=now  ∧
                T'<T.

Y W X -> ∀T:     holdsAt(Y, T) →
                  ¬holdsAt(X, T)
                  T≤T'<T'' ∧ T=now.
```

Figure 7: Event Calculus representation of temporal logic operators

The Event Calculus representation for each of these temporal operators is shown in Figure 7.

The UML profile in [9] also describes a high-level notation for representing these patterns, each of which can be mapped into a set of temporal logic formulas. These can be used by our system to guide the user in defining the sub-goals for a given goal and also to validate the correctness of the sub-goals. For example, applying the `(P => ◇ R, R => ◇ Q) ⊢ P => ◇ Q` pattern would present the user with a template of the following form:

```
If P is true then eventually R is true, AND
If R is true then eventually Q is true.
```

It would be up to the user to insert the appropriate value for the missing goal, R. The formal version of the goals would then be mapped into Event Calculus and the system would assert each of the sub-goals into the overall formal specification and attempt to prove the following properties of the refinement:

1. `G1, G2, .. Gn ⊢ G` *(entailment)*: validated by trying prove G after asserting all the sub-goals
2. `∀i: ∧ji Gj ⊬ G` *(minimality)*: validated by checking the entailment property for each subset of the sub-goals.

3. `G1, G2, …, Gn ⊬ false` *(consistency)*: validated by making sure that asserting the sub-goals does not nullify the entailment properties of any existing goal refinements.

If it is not possible to show the entailment property for the goal refinement, this indicates that either there is a missing sub-goal or the wrong goal refinement pattern has been applied.

3.3 Strategies

So far we have discussed the types of information that must be specified by the user for the refinement procedure to work. However, strategies do not fall into this category since they will be actually be derived by the abductive analysis procedure used in the refinement approach. Therefore, it is expected that the formal representation of a strategy is actually determined by the representation of the system behaviour and goals defined above.

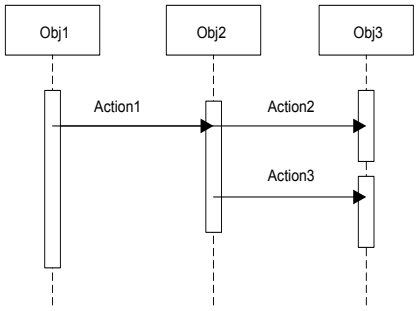

Figure 8: UML sequence chart for a strategy

As mentioned previously, a strategy describes the mechanism by which the system can achieve a given goal and is therefore defined by a set of operations to be performed sequentially or in parallel. Specifically, the strategy is defined using a conjunction of `happens(doAction(...), T)` predicates having a relationship between the time values that corresponds to the order in which the actions should be performed. For example, a strategy that defines `Obj1` performs `Obj2.Action1` and `Obj2` perfoms `Obj3.Action2` in parallel, followed by `Obj2` performing `Obj3.Action3` would be represented in our formalism as:

```
happens(doAction(Obj1, op(Obj2, Action1)), T0),
happens(doAction(Obj1, op(Obj3, Action2)), T0),
happens(doAction(Obj2, op(Obj3, Action3)), T1),
T0<T1.
```

In the interests of usability, it would be better if strategies are presented to the user in a high-level form. So, given that strategies define a method invocation trace for achieving a given goal, we can represent them in UML using a message sequence chart. The UML model for the example above is shown in Figure 8.

A strategy is considered to be abstract if any of the actions defined in it is a method defined as part of an abstract entity. High-level, abstract policies can be defined using such strategies in the action clause. If the strategy is not abstract, it can be used in a concrete, implementable policy.

4. Policy Refinement: An Example

In this section we describe how the formal representation and approach presented in this paper can be used to refine Service Level Agreement policies for the example system shown in Figure 1. Figure 9a shows the UML model for the objects in this system, including the abstract entities, `Network` and `Router`. The behavioural model is as shown in Figure 5. The high level policy we wish to refine is stated as follows:

> On demand the network should provide Gold quality of service to web services application traffic on the eCommerce server.

The goal we are interested in achieving is to provide gold QoS for network traffic to a particular application on the eCommerce server. The goal hierarchy for reducing this goal is shown in Figure 9b and the temporal logic representation for some of these sub-goals is presented below (tfc1 denotes the Traffic Class relevant to the goal):

```
G1  - send(pkt, tfc1) ⇒ ◇qos(pkt,gold).

G11 - send(pkt, tfc1) ⇒
            ◇routed(pkt, R, tfc1).

G12 - routed(pkt, R, tfc1) ⇒
            ◇detected(pkt, R, tfc1).

G13 - detected(pkt, R, tfc1) ⇒
            ◇configured(R, tfc1, gold).

G14 - configured(R, tfc1, gold) ⇒
            ◇qos(pkt, gold).

G131 - detected(pkt, R, tfc1) ⇒
            ◇routerParmsKnown(R, gold, parms).

G132 - routerParmsKnown(R, gold, parms) ⇒
            ◇ parmsSet(R, gold, parms).

G133 - parmsSet(R, gold, parms) ⇒
            ◇ configured(R, tfc1, gold).
```

At each level of goal reduction, we use abduction to determine the strategy that will achieve the sub-goals. The absence of a strategy indicates that there is some information missing in the system description at one of

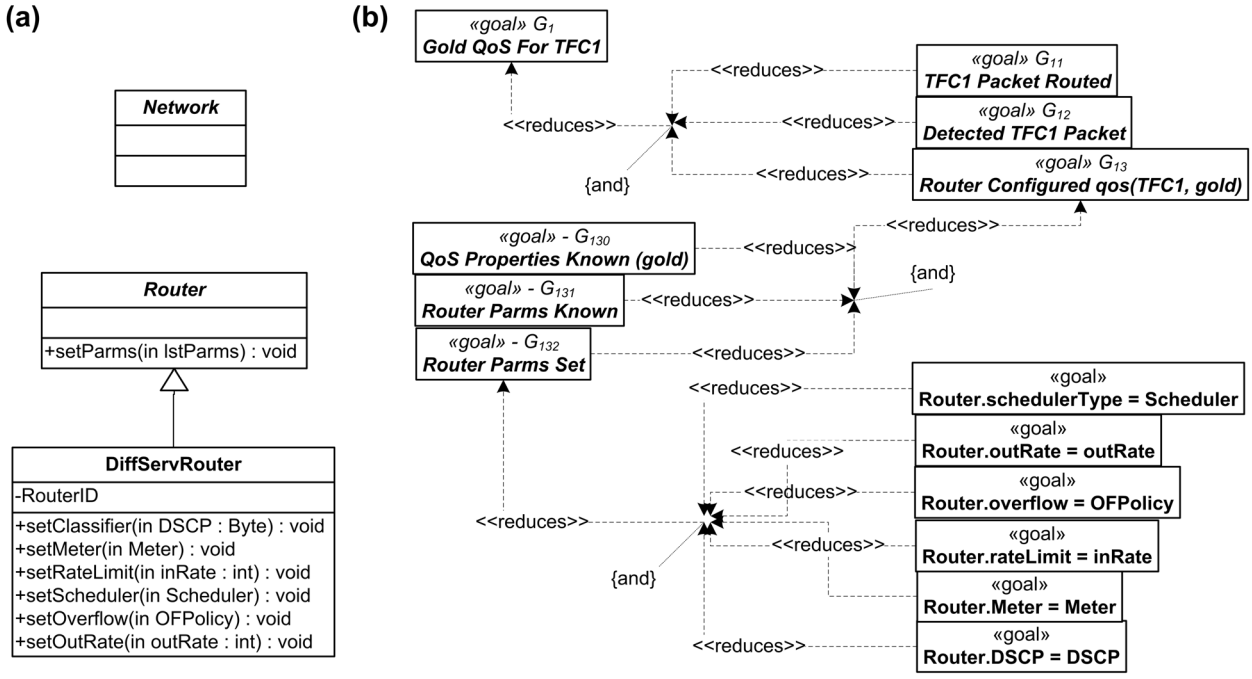

Figure 9: (a) UML representation of enterprise network example; (b) UML representation of goal hierarchy

the levels of abstraction. For example, at the top level of this example, there is no abducible strategy for the goal "G_{13} - Router Configured for Gold Qos". This can be addressed by extending the abstract `Router` object with a method `configureQoS(gold)`. Similarly, the strategy for the lower level goal, "G_{132} - Set the router parameters" can be achieved by defining the behaviour of the `setParms(…)` method of the `Router` object appropriately. Once these modifications have been made, the abduction process will yield abstract strategies (since the operations derived belong to abstract entities) for achieving each of the goals. In order to realise a concrete strategy, it is necessary to refine the goals further, into the lowest level ones shown in Figure 9b.

Now, attempting to abduce the lowest level goals yields a set of concrete operations that configure the `DiffServRouter` object in the appropriate way:

```
?- showStrategy([
    state(diffServRouter, dscp, Var_DSCP),
    state(diffServRouter, meter, Var_Meter), ..
    state(diffServRouter, ofp, Var_OFP]).

 1 - happens(doAction(_, diffServRouter,
                      setDSCP(Var_DSCP), 0),
 2 - happens(doAction(_, diffServRouter,
                      setMeter(Var_Meter), 0),
     ...
 6 - happens(doAction(_, diffServRouter,
                      setOverflow(Var_OFP), 2).
```

Having identified the actions required in the lower level policy, all that remains is to refine the subject and target entities. In the original high-level policy we can identify the target entity as "the network". For the policy we are refining, we are only concerned with objects that are of type `DiffServRouter` (since this is the only object type in the policy's action clause). Therefore the refined target objects can be determined by traversing the domain hierarchy and selecting the objects of type `DiffServRouter`.

For ease of future specification, we can create a new domain `DiffServRouters`, and assign each of these target objects as members. Given there is no information about the subject entity in the top-level policy, it requires the user to apply some application specific knowledge to identify the correct subject for the low-level policy as `DiffServConfigMgr`. The event mentioned in the high-level policy is "on Demand" and given that there is no information in the system description about how this might be refined; it is up to the user to specify the lower level event to be used by the policy as `adminRequest(Parms)`. This yields the final low-level policy as:

```
oblig /SLA/ConfigGoldQoS {
  on       adminRequest(Parms);
  subject  s = /PMA/DiffServeConfigMgr;
  target   t = /DiffDServRouters/;
  do       (t.setDSCP(Parms.DSCP) &&
            t.setMeter(Parms.Meter)) ->
            ...
            t.setOverflow(Parms.OFP);
}
```

5. Discussion

The approach described above provides a means of determining the strategy for achieving a particular goal, and identifying the specific objects in the system that are required to execute the strategy. However, there is no mention of how to decide whether a particular strategy should be specified as a policy, as opposed to directly implementing as system functionality. Using a policy specification differs from a direct implementation in that a policy controls the required functionality rather than implementing it directly. Therefore policies provide a great deal of flexibility in situations where there a several alternative strategies for achieving a goal, and it would be useful to dynamically switch between these strategies depending on the run-time state of the system. For instance, in the example scenario outlined above, the Gold QoS requirement might be met by either configuring the DiffServRouters in the manner described, or by dropping packets belonging to other applications. In this situation, two alternative low-level policies could be defined such that the strategy most appropriate for a given situation is used.

The exact circumstance in which a strategy should be encoded as a policy, rather than system functionality, will depend on the particular application domain. So, whilst there is no obvious way to automate this decision, we propose the following guidelines to determine the situations in which a policy-based implementation would be appropriate:

1. If the goal refinement process 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.
3. If a strategy has parameter values that the user is interested in changing at a future point in time, implementing such a strategy in a policy will provide the necessary flexibility to do this.

These guidelines should apply to all types of application. Additionally there may be application-specific guidelines that further guide the user in their decision to apply policies.

The policy refinement process described in this paper is built on a systematic, formal approach to refining goals thus ensuring that the strategies derived actually meet the requirements of the high-level policy. Also, the derivation of these strategies makes use of a description of the system, which means the policies derived are enforceable by the system. Using domain hierarchies to model the relationships between abstract entities and concrete objects, together with type information, allows the system to identify the objects that may be required to execute the strategies. These features illustrate how this solution satisfies the principal objectives of a policy refinement process identified in [1]. Additionally, by implementing the process using a formal representation it is possible to automate parts of the refinement process.

Automation of the technique presented here requires a tool that allows the user to specify the system behaviour and goal information in UML and then translates 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 envisage the final tool solution will integrate a UML editor, such as ArgoUML, with a Prolog system implementing an abductive reasoning engine. For the latter part of the solution, we will use the A-System with SICStus Prolog [11]. This latter part of the architecture has already been used to develop the policy analysis approach presented previously [5]. It is expected that this refinement and analysis tool will be integrated with a policy management system such as the Ponder Toolkit [8]. Development of an integrated refinement and analysis toolkit will form the core of our future work.

An important consideration when developing any formal technique is to ensure that the implementation is decidable and computationally feasible. In the Prolog implementation of the example, we have been able to ensure 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 [12]. It is anticipated that we can remain within the realms of stratified logic programs for most applications of our technique. This would be advantageous since there are numerous studies that identify stratified logic as a class of first order logic that supports logic programs that are decidable [13, 14]. Moreover, such programs are decidable in polynomial time [14, 15]. A more detailed analysis of the computational complexity and expressive power of stratified logic can be found in [14].

One limitation of the work presented is that it does not provide a means of deriving the parameter values required by the operations to achieve a particular goal. Such a capability would be particularly useful when refining network management policies, where for example there might be a requirement that the network configure itself to provide optimal bandwidth utilisation by calculating the appropriate values for parameters like the input rate of the DiffServ meters. As part of our ongoing research, we plan to investigate the possibility of integrating constraint

logic programming techniques to provide such capabilities. Another limitation is that at present we treat all the goals together, only accounting for whether their decomposition is based on the AND/OR connective. However, there may be situations where it is necessary to account for an explicit temporal ordering of the goals when performing refinement. Whilst this may be easily handled by making use of the time information provided by the Event Calculus representation, the implications must be fully considered and this requires further investigation.

6. Related Work

Work by Kelly [17], introduces the idea of annotating a goal refinement hierarchy with strategies for representing safety cases. However, in the context of safety case representation the strategies document the justification for the lower-level goals achieving the high-level goal. In contrast, the goal refinement approach used in this paper uses logical proofs to justify the validity of the goal decomposition and strategies are used to represent the mechanism by which the system can achieve a given goal. Therefore strategies provide the relationship between the system architecture and the goals.

In the wider software engineering context, there is a body of work on the synthesis of reactive systems [18], which aims to derive the system behaviour description based on temporal formulae that describe the output of the system. This is quite different from the approach presented in this paper, since our objective is to simply identify the sequences of actions, from the given system description, that will achieve a particular goal.

There are few examples of practical approaches for policy refinement. One such example is described in work done at Hewlett-Packard Laboratories, which outlines a policy-authoring environment that provides a policy wizard tool, called POWER, for refining policies [19]. Here, a domain expert first develops a set of policy templates, expressed as Prolog programs, and the policy authoring tools have an integrated inference engine that interprets these programs to guide the user through the refinement process. A major limitation of this approach is the absence of any analysis capabilities to evaluate the consistency of the refined policies. Also, the POWER approach depends on the domain expert having a detailed understanding of the entire system to develop a usable policy template. The refinement approach outlined in this paper avoids these problems by not only incorporating a complete analysis technique but also supporting abductive reasoning for deriving the action sequences required to achieve a goal.

7. Conclusions

In this paper we have presented an approach to policy refinement that allows the inference of the low-level actions that satisfy a high-level goal by making use of existing techniques in goal-based requirements elaboration and the Event Calculus. We have ensured the usability of the approach by showing how the user can specify the system using UML and how this specification can be translated into the formal representation for analysis. We have shown how the approach provides automation support for the refinement process when given a specification of the system behaviour and the goals to be satisfied. In order to relate the system behaviour specification with the goals, we introduce the concept of strategies and show how these can be used in the specification of policies.

There is ongoing work to investigate how the presented formalism can be extended to support the identification of the events and constraints to be included in the low-level policies. However, the immediate focus of our future work is to develop adequate tool support that uses the technique described here together with the analysis approach presented previously [3] to provide a comprehensive environment for policy specification. Additionally we will be investigating the use of the technique described here for refining and analysing traffic management policies for network QoS management. The areas of further investigation identified in the discussion will also be addressed as part of this work.

Acknowledgements

We acknowledge financial support for this work from the EPSRC (Grant Nos: GR/R31409/01 and GR/S79985/01) and CISCO Systems Inc. Additionally, we would like to thank Morris Sloman and Naranker Dulay for their valuable feedback during the preparation of this paper.

References

[1] J. Moffett and M. S. Sloman, "Policy Hierarchies for Distributed Systems Management," *IEEE JSAC*, vol. 11, pp. 1404-14, 1993.

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

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

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

[6] R. Miller and M. Shanahan, "The Event Calculus in Classical Logic Alternative Axiomatisations," in *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II.*, vol. 2048, *Lecture Notes in Computer Science*, A. Kakas and F. Sadri, Eds.: Springer, 1999, pp. 452-490.
[7] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer, "An Abductive Approach for Analysing Event-Based Requirements Specifications," presented at 18th Int. Conf. on Logic Programming (ICLP), Copenhagen, Denmark, 2002.
[8] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman, "Tools for Domain-based Policy Management of Distributed Systems," presented at Network Operations and Management Symposium (NOMS 2002), Frorence, Italy, 2002a.
[9] W. J. Heaven and A. Finkelstein, "A UML Profile to Support Requirements Engineering with KAOS," *IEE Proceedings - Software*, 2003.
[10] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*: Springer-Verlag, 1992.
[11] B. van Nuffelen and A. Kakas, "A-System : Programming with abduction," presented at Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), 2001.
[12] K. R. Apt, H. A. Blair, and A. Walker, "Towards a Theory of Declarative Knowledge," in *Foundations of Deductive Databases*, J. Minker, Ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 89-148.
[13] G. Jager and R. F. Stark, "The Defining Power of Stratified and Hierarchical Logic Programs," *Journal of Logic Programming*, vol. 15, pp. 55-77, 1993.
[14] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and Expressive Power of Logic Programming," presented at 12th Annual IEEE Conf. on Computational Complexity (CCC'97), Ulm, Germany, 1997.
[15] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A Logical Language for Expressing Authorisations," presented at IEEE Symposium on Security and Privacy, Oakland, USA, 1997a.
[16] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner, "Nonmonotonic causal theories," *Artificial Intelligence*, vol. To appear, 2003.
[17] T. Kelly, "Arguing Safety – A Systematic Approach to Managing Safety Cases," in *Department of Computer Science*. York: University of York, 1998, pp. 341.
[18] A. Pneuli and R. Rosner, "On the Synthesis of a Reactive Module," presented at ACM Symposium on the Principles of Programming Languages (POPL'89), pp. 179-190, 1989.
[19] M. Casassa Mont, A. Baldwin, and C. Goh, "POWER Prototype: Towards Integrated Policy-Based Management," HP Laboratories Bristol, Bristol, UK October 1999.