Reasoning Techniques for Analysis and Refinement of Policies for Service Management

Antonis Kakas, Arosha K Bandara, Alessandra Russo, Emil C Lupu, Morris Sloman, Naranker Dulay

Technical Report 2005/7
Department of Computing, Imperial College London

Abstract

The work described in this technical report falls under the general problem of developing methods that would allow us to engineer software systems that are reliable and would offer a certain acceptable level of quality in their operation. This report shows how the analysis and refinement of policies for Quality of Service can be carried out within logic by exploiting forms of abductive and argumentative reasoning. In particular, it provides two main contributions. The first is an extension of earlier work on the use of abductive reasoning for automatic policy refinement by exploiting the use of integrity constraints within abduction and its integration with constraint solving. This has allowed us to enhance this refinement process in various ways, e.g. supporting parameter values derivation to quantify abstract refinement to specific policies ready to be put in operation, and calculating utility values to determine optimal refined policies. The second contribution is a new approach for modelling and formulating Quality of Service policies, and more general policies for software requirements, as preference policies within logical frameworks of argumentation. This is shown to be a flexible and declarative approach to the analysis of such policies through high-level semantic queries of argumentation, demonstrated here for the particular case of network firewall policies where the logical framework of argumentation allows us to detect anomalies in the firewalls and facilitates the process of their resolution. To our knowledge this is the first time that the link between argumentation and the specification and analysis of requirement policies has been studied.

This deliverable describes the work carried out under the visiting fellowship of Professor Kakas at Imperial College, London. This fellowship was supported by EPSRC (grant GR/T29246/01) under the project "Reasoning Techniques for Analysis and Refinement of Policies for Quality of Service Management".

1. Introduction

A key issue in policy-based approaches for systems management is the development of methods for analysis and refinement of policy specifications, in particular within the context of policies applicable across administrative domain boundaries. Such specifications often include authorisation, management policies, and general and/or application-specific constraints. The ability to detect domain-independent and application-specific conflicts among policies, identify the causes for these conflicts and propose resolutions is crucial. However, most of the research effort on policy-based system management has been mainly focused on the development of languages for specifying policies and architectures for managing and deploying policies in distributed environments. Existing analysis techniques for policies specifications have been limited to the study of modality and application-specific conflicts. In the last few years, the investigators have successfully shown how abductive reasoning, combined with an Event-Calculus formalism for specifying policies, can be used to analyse policies, detect different types of conflicts and generate explanations in terms of the conditions under which such conflicts arise [1]. This logic-based approach has been effectively applied in the context of Quality of Service (QoS) provisioning in Differentiated Services (DiffServ) networks for the detection of policy conflicts during network dimensioning (i.e. static resource management of QoS provisioning) [6]. *The first part of this report builds upon this results and shows how to integrate argumentation methods with abductive reasoning in order to enable more modular modeling and analysis techniques for policy specifications and support resolution of policy conflicts.*

A second critical task in policy-based specifications is the derivation of policies from high-level requirement specifications. Within the specific context of network service management, QoS policies would need to be derived from given Service Level Agreement (SLA) specifications and provider's business goals in order to provide policy management rules for dynamically reconfiguring the routers in a network that enable the desired QoS goals to be achieved. Such policies are normally derived manually with no means of verifying their correctness with respect to the underlying network devices and the desired QoS goals. The process of deriving policies from the SLA specifications in a fully automated way is recognized as a difficult research challenge. However, tool support to assist human administrators in the refinement of policies from SLAs and enterprise goals would significantly reduce and improve network administration tasks especially when combined with analysis tools to ensure that only consistent specifications are derived. Policy refinement would in this case require mapping abstract entities, defined as part of high-level requirements, to concrete objects/devices of the underlying system, defining concrete policies in terms of only operations supported by the underlying system, and ensuring that at each stage of refinement the decomposition of abstract requirements into more concrete policies is correct and consistent [5; 6]. Although policy refinement cannot be fully automated in general, increasing support for policy refinement can be achieved when constraining the problem to a well-defined functional area, such as QoS management, where application specific knowledge can be encoded and used. In particular in [6] we have shown that abductive reasoning can be used to derive sequences of policy-based operations (i.e. *strategies*) that would allow a given system to achieve its high-level desired goal. This provides an initial formal approach and tool support for *partial* policy refinement, in which the derived strategies can be then used to manually specifying policies that can be enforced by the system. *The second part of this report shows how to fully exploit abduction constraint solving to increase automated support for policy refinements.*

PART A: <u>Software Requirements as Argumentation based Preference Policies</u>

The need to use formal techniques for software engineering specifications is now well proven. In particular, several studies have shown how formal logic can be used to formalize and analyse software requirements. One such work is [1], which investigates the use of abductive reasoning in requirements *policy analysis and refinement*, to address the issues of conflict detection and validation. This work has shown how policy analysis can be facilitated greatly when policies are expressed in a high-level declarative representation framework such as that of Abductive Logic Programming [7, 8, 11, 29] with Event Calculus [5, 6] logical formalization.

Policy-based approaches to network and system management are particularly important because they allow separation of the rules that govern the behavior of a system from the functionality provided by that system. Policies allow the behavior of a system to be adapted, without the need to recode its functionalities. They capture the relative priority or preference amongst the different possible actions that the system can take at any stage in its operation. Expressing policy specifications as *preference policies* provides us with a higher level of abstraction and greater degree of flexibility in their formulation and analysis. Indeed, in many cases software requirements specifications can be seen naturally as policies of the preferred way of operation that the software systems need to follow. For example, we may have requirements policies for quality of service or for the security level provided by our software systems. Understanding software requirements as preference policies allows us to view these at a higher-level of abstraction and as a result of this to have a greater degree of flexibility in the allowed formulation and analysis of the requirements. The modularity of such high-level representations can be exploited to facilitate the development of our policies.

Within this context, we have investigated how to model and formulate QoS policies, and other policy-based systems managements, as preference policies within logical frameworks where the preference reasoning is realised via ***argumentation***, a form of reasoning closely relation to abduction, using in particular the framework of *Logic Programming with Priorities* [27, 28]. We have examined how we can exploit the *modularity* of such high-level logical representations to facilitate the editing and updating of given policies. We have explored how argumentative reasoning and its combined use with abductive reasoning, can give us a framework for *high-level abstract* analysis of policy specifications through the use of semantic queries based on these forms of reasoning. We have then investigated the benefits of this approach by applying it to the context of firewalls policies for network security.

In what follows we will explain our approach and present its main properties within the context of firewall policies, but we claim that these are valid more generally for other application domains of policy-based system managements. To our knowledge this is the first time that the link between argumentation and the specification and analysis of software requirements is proposed and explored.

Section 2.1 gives the necessary background on argumentation and the framework of LPP employed in our study. It is then demonstrated how software requirements policies can be represented in this framework and discussed a methodology for developing such policies within this argumentation-based framework. Section 2.2 presents an extensive study of the application of these ideas to the problem of modelling, analyzing and developing network security policies.

A.1 Argumentation Basics

Argumentation has been shown to be a useful framework for formalizing non-monotonic reasoning and other forms of reasoning (e.g. [33,34,35,36,]). In general, an argumentation framework is a pair <T,*A*> where T is a theory in some background (monotonic) logic, equipped with an entailment relation, $\models$, and *A* is a binary relation on the subsets of T. These subsets of T form the *arguments* of the framework and *A* is therefore an *attacking relation* between arguments. For any two arguments A1 and A2 we say that *A1 attacks A2* when (A1,A2) belongs to the attacking relation *A*.

The semantics of an argumentation framework is based upon the following central notion of an *admissible* argument.

Definition (Admissibility of Arguments)
Given an argumentation framework <T,*A*>, an argument, Δ, is **admissible** if and only if

1. Δ does not attack itself,
2. for all arguments Δ', if Δ' attacks Δ then Δ also (counter-)attacks Δ'. ◊

We will use a particular framework of argumentation as realized by the framework of Logic programming with Priorities (LPP) and its concrete form of *Logic Programming without Negation as Failure* (LPwNF).

The preference reasoning within LPwNF is based on a model of argumentation where local priority information, given at the level of the rules of a theory (or policy), is lifted to give a global preference over sets of rules that compose arguments and counter arguments for a certain decision. A theory or policy within LPwNF is viewed as a pool of sentences or rules from which we need to select a suitable subset, i.e. an argument, in order to support a conclusion.

In LPwNF a theory or policy, T, is represented in a classical background logic (*L*, $\models$) where the language *L* consists of (Extended) logic programming rules of the form:

$$\text{Name} : L \leftarrow L_1, \ldots, L_n, \ (n \geq 0)$$

Here, $L, L_1, \ldots, L_n$ are positive or negative (classical) literals. A negative literal is a literal of the form ¬A, where A is an atom. As usual in Logic Programming a rule containing variables is a compact representation of all the ground rules obtained from this under the Hebrand universe. Each ground rule has a unique (parametric) name, Name, given at the front of the rule. The background entailment relation, $\models$, of LPwNF is that of monotonic Horn logic given by the single inference rule of Modus Ponens, treating negative literals as ordinary atoms.

In general, we can separate out an auxiliary part, T_0, of a given theory, T, from which the other rules can draw background information in order to satisfy some of their conditions. The reasoning of the auxiliary part of a theory is independent of the main argumentation-based preference reasoning of the framework and hence any appropriate logic can be used. The auxiliary part contains also the definition of what constitutes a conflict in our theory (over and above the standard conflict of classical negation, i.e. between an atom, A and its negation ¬A). This is given through the definition of an auxiliary predicate, incompatible/2, of the form

$$\text{incompatible}(L_1, L_2) \leftarrow B$$

stating that literals L_1 and L_2 are conflicting under some (auxiliary) conditions B. Typically, the conditions B are empty and the definition of the incompatible predicate is kept simple.

The non-auxiliary part of a theory T, which encodes the proper policy part of our theory, is separated into two parts: the *basic* part and the *strategy* part. The basic part contains rules (of the form given above) whose conclusions, L, are any literal except the special predicate, priority/2, which is the only predicate that can appear in the conclusion of rules in the strategy part. Hence rules in the strategy part take the special form.

$$\text{Name} : \text{priority}(\text{rule1}, \text{rule2}) \leftarrow L_1, \ldots, L_n, (n \geq 0)$$

where rule1 and rule2 are the names of any other two rules in the theory.

A rule of this form then means that under the conditions $L_1, \ldots, Ln$, the rule with name, rule1, has priority over the rule with name, rule2. The role of this priority relation is therefore to encode locally the relative strength of (argument) rules in the theory. The priority relation priority/2 is required to be *irreflexive*. The rules rule1 and rule2 can in fact be themselves rules expressing priority between other rules and hence the framework allows higher-order priorities. For simplicity we will assume that the conditions of any rule in the theory do not refer to the predicate priority/2 thus avoiding self-reference problems. Note also that the definition of incompatible/2 always includes that any ground atom, priority(rule1, rule2), is incompatible with the atom priority(rule2, rule1) and vice-versa.

The attacking relation, *A*, of LPwNF is then given by combining together the conflicts and priority relations in our given theory, T. For any two sets Δ, Δ' of sentences in T, Δ **attacks Δ'**, whenever these two sets have some conflicting (i.e. incompatible) conclusion (under the background logic $\models_H$) and that the rules of Δ that derive this are rendered by Δ to have at least the same priority as the priority that Δ' renders for its own rules that derive the conflicting conclusion. This is formalized by the following definition.

Definition (Attacking Relation of Arguments in LPwNF)
Let T be an LPwNF theory and Δ, Δ' subsets of T. Then Δ' attacks Δ (or Δ' is a counter argument of Δ) if and only if there exists a literal, L, and subsets Δ_1 of Δ' and Δ_2 of Δ such that:

(i) $\Delta_1 \models L$ and $\Delta_2 \models LC$, minimally
(ii) $(\exists r \in \Delta_1, s \in \Delta_2 \text{ s.t. } \Delta_2 \models \text{priority}(s, r)) \Rightarrow$
$(\exists r \in \Delta_1, s \in \Delta_2 \text{ s.t. } \Delta_1 \models \text{priority}(r, s))$

where LC is any literal that conflicts L (e.g. LC = ¬L or incompatible(L,LC) holds) and, $\Delta \models L$ minimally, means that $\Delta \models L$ and that L cannot be derived from any proper subset of Δ. ◊

The second condition in the above definition states that an argument Δ' for L attacks an argument Δ for the contrary conclusion only if the set of rules that it uses to prove L are at least of the same strength (under the priority relation priority/2) as the set of rules in Δ used to prove the contrary. Note that the attacking relation is typically not symmetric.

This notion of attack, that lifts the priority relation from the individual rules to sets of rules, then gives us the admissible arguments of any LPwNF theory according to the above general definition of admissibility.

Preference reasoning is based on the *maximal admissible* arguments of a given theory. Usually, two *preference entailment relations* are defined.

Definition (Preference Entailment in LPwNF)
Given an LPwNF theory, T, and an atomic goal, G, we define a *credulous* and a *sceptical preference entailment* as follows:

- T $\models_{cpr}$ G means that there is at least one maximal admissible subset of T where G holds under the background logic $\models$;
- T $\models_{spr}$ G means that T $\models_{cpr}$ G and, that for any GC such that incompatible(GC,G) holds, T $\models_{cpr}$ GC does not hold. ◊

We can easily extend this definition, in the obvious way, for goals that are a conjunction of literals.

Hence in the case of a credulous conclusion of our theory or policy we know that there is at least one admissible argument that is supporting it but there could also be other admissible arguments for conflicting conclusions. The conclusion is possible. When no conflicting conclusion has an admissible argument for it we have that the conclusion is sceptically entailed by our theory. The conclusion is certain.

Given a policy theory, T, we can identify three important properties that this policy may have. The first one is the property of *determinism* in its decision, i.e. that the policy always has only one sceptically preferred conclusion under any specific background situation that can arise in the problem domain for which the policy is to be applied. Equivalently, this means that there is an admissible argument for only one conclusion of the policy. In the opposite case we have non-determinism where the policy can support different incompatible conclusions for the same situation at hand. The other important property of a policy theory is that of *non-redundancy*, i.e. that for every subset, S, of the basic part of our theory T, there exists at least one background situation, described by an auxiliary theory, S_0, under which S belongs to at least one admissible subset of T ∪ S_0. Hence we do not have parts of our policy that would not apply ever. Finally, the third important property is that of *exhaustiveness*, i.e. that for any background situation, described by an auxiliary theory, S_0, the theory T ∪ S_0 has at least one admissible subset that supports a conclusion on some desired goal predicate.

A.2 Software Requirements as Argumentation theories

In this section we will illustrate the framework of LPwNF and how its argumentation semantics allows us to carry out preference reasoning through a “standard” example of a requirements policy. We will show here how we can formulate and analyse within LPwNF an Elevator Policy that specifies the (preferred) mode of operation of an elevator. The task of this policy is to specify what action the elevator should take at any given instance of its operation. The available actions for the elevator are:

- open_door : Elevator doors open
- go_up : Elevator goes up one floor
- go_down : Elevator goes down one floor

The “world situations” of operation are parameterized by the actions executed externally by the elevator users. These actions are:

- press_button_in(Floor) : A user presses a button from inside the elevator for the floor, Floor.
- press_button_floor(Floor, Direction) : A user presses a button at Floor for Direction – either up or down.

These two sets of actions change the state of the world. These states are described using the following auxiliary fluent predicates:

- elevator_position(Floor): Elevator is at Floor
- activated_button_in(Floor) : The button for Floor inside the elevator is activated
- activated_button_floor(Floor, Direction): The button at Floor for Direction is activated

We reason about these fluents and their changes using the Event Calculus (EC), assumed to be part of our background auxiliary theory, T_0. Therefore, T_0 contains the standard domain independent axioms of the EC

- holds(P,T) ← holds-initially(P),T ≥ 0,not broken(P,0,T).
- holds(P,T) ← happens(A,T'), T'<T, initiates(A,T',P), not broken(P,T',T).
- broken(P,T',T) ← happens(A,T''), terminates(A,T'',P), T'<T''<T

together with domain dependent axioms such as:

% When a button is pressed it becomes activated:
- initiates(happens(press_button_in(Floor)), T, activated_button_in(Floor)).

% When the elevator arrives at a floor the buttons pressed for that floor stop being
% activated:
- terminates(action(open_door,T), T,activated_button_in(Floor)) ←
  holds(elevator_position(Floor), T), holds(activated_button_in(Floor), T).

Similar domain dependent statements, about the initiating and terminating effects of all the actions, are included in the background theory T_0 and can be found in appendix A, where the full elevator policy as a LPwNF theory is given, written in the GORGIAS system that implements the framework of LPwNF.

The background theory T_0 also contains the following single statement of incompatibility that expresses that any two different actions of the lift are incompatible, or in other words that the lift can decide to execute only one action at a time:

% Different actions are incompatible with each other:
- ¬ action(A, T) ← action(B, T), A ≠ B

We are now ready to represent the main part of the policy theory of the elevator, namely the rules that specify (generate) under some conditions the possible actions that the elevator can take in the basic part of the policy and the relative priority rules on these generation rules and actions in the strategy part of the policy.

The basic part of the policy theory T contains rules of the following form.

% The elevator can go up one floor when a button is inside the elevator is activated for a % floor higher than the elevator's current position:
- r_{up_in}(Floor1, Floor2):
```
    action(go_up, T) ←
       holds(elevator_position(Floor1), T),
       holds(activated_button_in(Floor2), T),
       Floor2 > Floor1.
```

% The elevator can go up one floor when a button is at a floor higher that the elevator's % current position is activated:
- r_{up_floor}(Floor1, Floor2):
```
    action(go_up, T) ←
       holds(elevator_position(Floor1), T),
       holds(activated_button_floor(Floor2,Dir), T),
       Floor2 > Floor1.
```

Similar rules exist for the action to go down. For the action to open the doors we have rules of the form:

% The elevator' doors may open when the elevator is at a floor and a button for that % floor is activated inside the elevator:

- r_{open_in} (Floor):
  action(open_door, T) ←
  holds(elevator_position(Floor), T),
  holds(activated_button_in(Floor), T).

Turning now to the strategy part we need to specify the priority that our policy should have when we have situations where at the same time we have buttons activated both above and below the lifts current position. In such situations more than one of the rules in the basic part of the policy "fires" and hence we need to specify the preferred action amongst the many actions possible.

Below we give some example priority rules in the strategy part. As mentioned above a full policy can be found in appendix A that is executable in the GORGIAS system. When writing these rules we exploit the fact that our rules have names and therefore we can refer directly to the specific rules and situations of interest when we are specifying priorities amongst possible actions.

% The elevator' doors should not open (when a button is activated at a floor for a % certain direction) if the elevator is moving in the opposite direction

- $R_{down\_in|open\_floor}$:
  r_{down_in}(Floor1, Floor2) > r_{open_floor}(Floor1, up)
- $R_{down\_floor|open\_floor}$:
  r_{down_floor}(Floor1, Floor2) > r_{open_floor}(Floor1, up)
- $R_{up\_in|open\_floor}$:
  r_{up_in}(Floor1, Floor2) > r_{open_floor}(Floor1, down)
- $R_{up\_floor|open\_floor}$:
  r_{up_floor}(Floor1, Floor2) > r_{open_floor}(Floor1, down)

Note that in the presentation of these rules we are using instead of the priority/2 predicate the infix notation ">".

Similarly, we have the following strategy rules that give priority to the action of opening the lift doors.

% Elevator doors should open when the elevator is at a floor and the button of that floor % is activated from inside the elevator

- $R_{open\_in|up\_in}$:
  r_{open_in} (Floor1) > r_{up_in}(Floor1, Floor2)
- $R_{open\_in|up\_floor}$:
  r_{open_in} (Floor1) > r_{up_floor}(Floor1, Floor2)
- $R_{open\_in|down\_in}$:
  r_{open_in} (Floor1) > r_{down_in}(Floor1, Floor2)
- $R_{open\_in|down\_floor}$:
  r_{open_in} (Floor1) > r_{down_floor}(Floor1, Floor2

Note that these four rules could also be written more succinctly as a single rule:

- R(open_in,Action):
  r(open_in,Floor1) > r(Action,Floor2) ← Action≠open_in

where we are using variables to parameterize the names of our rules.

Another aspect of the strategy part of our policy could give priority to the activation requests from inside the elevator over those requests that come from outside the elevator at some floor. This is captured by the priority rules:

- Rup_in|down_floor:
  rup_in(Floor1, Floor2) > rdown_floor(Floor1, Floor3)
- Rdown_in|up_floor:
  rdown_in(Floor1, Floor2) > rup_floor(Floor1, Floor3)

At this stage we can enquire if the policy is deterministic, i.e. that it has a unique preferred choice at any situation. We see though that we have cases of conflict between our priority rules. An example is the following where on the one hand we are giving priority to opening the door at some floor over going down (due to the priority to open the doors when the lift is at a floor) and on the other hand we are giving priority of going down over opening the doors (due to the priority that the lift is moving in the opposite direction):

- Ropen_floor|down_in :
  ropen_floor (Floor1, Direction) > rdown_in(Floor1, Floor2)
- Rdown_in|open_floor:
  rdown_in(Floor1, Floor2) >ropen_floor(Floor1, up)

Our policy therefore contains a conflict now seen at the level of the priorities in the policy. We can resolve such conflicts by employing **higher-order** priority rules. In our example above we can ask the user what is the desired behaviour of the lift by explicitly asking what is meant to be stronger "the requirement to stop and open the doors at a floor where there is a request for the lift or the requirement to go down past a requested floor when the request is to go up". Suppose that the user answers that "carrying on past the requested floor is stronger" then we can represent this with the higher-order priority rules:

- $P_{down\_in|open\_floor}$:
  $R_{down\_in|open\_floor} > R_{open\_floor|down\_in}$
- $P_{down\_floor|open\_floor}$:
  $R_{down\_floor|open\_floor} > R_{open\_floor|down\_floor}$
- $P_{up\_in|open\_floor}$:
  $R_{up\_in|open\_floor} > R_{open\_floor|up\_in}$
- $P_{up\_floor|open\_floor}$:
  $R_{up\_floor|open\_floor} > R_{open\_floor|up\_floor}$

A.3 Conflict Detection and Resolution

As we have seen above some conflicts can be identified during the process of composing the policy simply by examining the rules of the policy and checking that there is a common situation in which conflicting conclusions can be derived. But once we are satisfied that we have captured the policy how can we check that it is in fact conflict free? Is it possible to detect in an automatic way the existence of conflicts in the policy?

To answer this we first need to notice that the existence of a conflict in LPwNF policies corresponds to the fact that the policy is non-deterministic, i.e. it has more than one admissible recommendation for a specific situation or scenario. So a conflict can be defined at the meta-level as:

- **conflict(Scenario)** if $T \cup$ Scenario $\models_{cpr}$ G1,
  $T \cup$ Scenario $\models_{cpr}$ G2,
  incompatible(G1,G2).

where Scenario describes a specific situation in which a conflict exists. Under such a conflicting scenario therefore we can get two incompatible conclusions. Hence to find such a scenario (if it exists) we can follow the following steps:

1. Query the policy for a certain recommendation, e.g. action(go_up, T) in the lift example.
2. Get the background scenario that would give this. This can be done by employing abduction in the query of step (1).
3. With this same background scenario query the policy for a different recommendation than the one in step (1) which is incompatible with it, e.g. action(go_down, T) in the lift example.

Let us see another example of a conflict in the elevator policy. A conflict scenario is given by:

- **Conflict Scenario**:
    - holds_initially(elevator_position(3))
    - happens(press_button_floor(1, up), T)
    - happens(press_button_floor(5, up), T)

- **Conflicting Recommendations & Rules:**
    - action(go_up,T) - r_{up_floor}(Floor1, Floor2)
    - action(go_down,T)- r_{down_floor}(Floor1, Floor3)

This shows that when we have at the same time a request from above and below the current position of the lift the policy is able to recommend either to go up or to go down. In other words, we somehow have forgotten to consider this scenario when formulating the policy.

To resolve this conflict we ask the user what is more important or stronger in this situation. Suppose the user replies that the elevator should move in the direction with the closest request. Then we can extend our policy with the following rule to capture this and thus resolve the conflict:

- $R_{up\_floor|down\_floor}$:
    r_{up_floor}(Floor1, Floor2) > r_{down_floor}(Floor1, Floor3)
        if abs(Floor2-Floor1) ≥ abs(Floor3-Floor1)
- $R_{down\_floor|up\_floor}$:
    r_{down_floor}(Floor1, Floor2) > r_{up_floor}(Floor1, Floor3)
        if abs(Floor2-Floor1) > abs(Floor3-Floor1)

Note the asymmetry in these two rules where in the first rule we have "≥" in the comparison of distances thus giving priority to going up over down when the distance of request from above and below is exactly the same.

In this way we can keep extending our policy in a modular and incremental way until it is conflict free. A conflict free policy is characterized by the following two conditions on the form of our policy rules:

- There exist priority rules that decide between any two rules that give conflicting actions in the same background scenario.

- There exist higher-order priority rules that decide between any conflicting priority rules in the same background scenario.

We could do an analogous analysis of our policy to check that it has the other two properties of non-redundancy and exhaustiveness. This leads us to the following **methodology** for developing a policy in the LPwNF framework:

1. **Compose** Policy rules in Basic and Strategy parts
2. **Analyse** Policy for the three main properties of determinism, non-redundancy and exhaustiveness.
3. **Resolve** (or address the lack of any one of) these properties by modular and incremental extensions of the policy guided by the user by presenting to her/him the problematic scenarios and the parts of the policy (arguments) that lead to the problem.

A.3.1 Elevator: Example of Operation

Let us illustrate here through an extensive example the use of the above elevator policy as a preference policy for deciding the action of the lift. Assume that initially we have the following situation:

- Elevator is at floor1 with doors closed
- No buttons activated inside or outside the elevator
- John is at floor 1 and has pressed the "up" button. He wants to go to the 3rd floor
- Mary is at floor 2 and presses the "down" button. She wants to go to the 1st floor.

Hence at time 1 the following fluents hold:

- elevator_position(1)
- activated_button_floor(1, up)

The only rule in the basic part of the policy that fires is the rule:

- r_{open_floor}(1,up) for the action open_door

and hence the only possible action is to open_door. Then

- Elevator Doors open
  - activated_button_floor(1, up) no longer holds
- John enters the elevator
- John presses the "Floor 3" button.

As a result now

- activated_button_in(3)holds

and the only rule in the basic part the fires is

- r_{up_in}(1, 3) for the action go_up

and hence the only possible action is to go_up. Then

- Elevator goes up one floor
  - elevator_position(1*)* no longer holds
  - elevator_position(2) holds

Mary at floor 2 has pressed the "down" button

- activated_button_floor(2, down) holds

Now the possible actions are:

- go_up (from rule $r_{up\_in}(2, 3)$)
- open_door (from rule $r_{open\_floor}(3, down)$)

The relevant priorities & priority rules are:

- $r_{up\_in}(2, 3) > r_{open\_floor}(3, down)$
  (rule $R_{up\_in\,|\,open\_floor}$)

- $r_{open\_floor}(3, down) > r_{up\_in}(2, 3)$
  (rule $R_{open\_floor\,|\,up\_in}$)

Higher-order priority rule:

- $R_{up\_in|open\_floor} > R_{open\_floor|up\_in}$
  (rule $P_{up\_in\,|\,open\_floor}$)

Hence the chosen action according to the policy is go_up, past Mary. The elevator goes up one floor.

- Elevator goes up one floor
  - elevator_position(2) no longer holds
  - elevator_position(3) holds
- Possible actions
  - go_down (rule $r_{down\_floor}(3, 2)$)
  - open_door (rule $r_{open\_in}(3)$)
- Priority rules
  - ropen_in(3) > rdown_floor(3, 2)
    (rule Ropen_in|down_floor)
- Choosen action: open_door

Then John exits the elevator, activated_button_in(3) no longer holds, and the only possible action is go_down since activated_button_floor(2,down) still holds that allows the basic part rule, $r_{down\_floor}(3, 2)$), to fire. The elevator will then go down, open the doors for Mary and then when Mary presses the Floor 1 button the elevator will go_down to the first floor.

A.4. Modeling Firewall Policies

Security is a major consideration that must be addressed in any modern enterprise network. Broadly security can be split into the areas of authentication, authorisation, integrity and availability. Authentication is concerned with verifying a user's identity; authorisation focuses on ensuring that users are only allowed to perform those operations for which they have been granted permission (or are not prohibited from doing); integrity refers to ensuring that data is not modified without authorisation; and finally availability addresses problems such as denial of service where access to a particular resource is denied even though the user is authorised.

Whilst some of these security considerations are addressed at the application level, in many cases the most common tool in the administrators' arsenal is the network firewall. Therefore, ensuring that the rules that govern the behaviour of the firewalls are consistent and correct with respect to the security requirements of the system is critical. However, in large scale networks with multiple firewalls and a range of user security requirements, managing the firewall policies quickly becomes a non-trivial task.

Network Security Policies are typically realized via Firewall Policies [37, 43] that control the traffic between two domains on the network. These firewall policies consist of a totally ordered set of rules of the form:

<order> : <action> if <network conditions>

where the <network conditions> identify a certain type of traffic, typically from one domain to another under some protocol, and the action field, <action>, typically takes the values "accept" or "deny" thus specifying if the traffic is to be allowed to flow or stopped. The semantics of the firewall policy is given operationally and it is crucially dependent on the total ordering of its rules. The ordering position of a rule is given by a (unique) natural number in <order>. Then for any given packet check the rules of the policy according to this priority order. The first rule whose network conditions are satisfied by the packet determines through its action field the fate of the packet. All other rules that are ordered lower are ignored.

An important shortcoming of the current methods of developing Firewall Policies is the lack of a systematic link between the high-level specification of the security policy requirements and the firewall policy that is meant to capture these requirements through its operation. For example, how can we capture the requirement that all traffic from a name server should be accepted when at the same time we require that traffic from the network in which the server is placed is to be restricted depending on the destination by some other policy. Similarly, how can we reconcile two opposing firewall policies both of which are in the path of a certain type of network traffic?

We will study how such legacy firewall policies can be lifted to a declarative preference policy expressed in the argumentation based framework of Logic Programming with Priorities presented in section A. Such a lifted representation is high-level and closer to the natural specification of security policy requirements. We will show how the standard declarative semantics of LPP captures the operational semantics of the legacy firewall policies and then examine various possibilities of extending the scope of these firewall policies beyond the current realm of applicability of firewall policies.

A.4.1 Legacy Firewall Policies

For the purposes of this report and without loss of generality we will confine ourselves to legacy firewall policies whose rules have the following form (see [37,43] for details on the syntax and table format of presentation of these rules):

<order> : <action> if <protocol ><src_ip><src_port><dst_ip><dst_port>

The network conditions are therefore a 5-tuple filter capturing conditions on any packet:

<protocol>	- on the protocol type
<src_ip>	- on the source IP address
<src_port>	- on the source port
<dst_ip>	- on the destination IP address
<dst_port>	- on the destination port

Conditions on IP addresses can be that this is a particular host (specified either as "192.112.1.1" or with a machine name "zeus") or that it belongs to a network (again specified either as "192.112.1.*" or with a network name "net_cs") or that it can be any (specified by "any") address. Ports can be required to be either a single specific port number or any port number, indicated by "any", or some range of port numbers. For any given firewall policy, FP, and specific packet, Packet, we will write

FP(« Packet: Action »)

to denote that the policy FP recommends the action, Action, for this packet, e.g. if packet, pac_13, is denied under FP we will write FP(« pac_13: deny ») and similarly we will write FP(« pac_7: accept») when FP recommends to accept packet pac_7.

It is clear that these network field conditions can be captured directly by binary auxiliary predicates in a background logical theory. For example, source IP conditions can be captured by the binary relation "src_ip(Packet, IPAddress)", e.g. we can have "src_ip(pac1,net_cs)" specifying that the packet "pac1" has a source IP address from the network "net_cs", or we can have "src_ip(pac2,any)" specifying that packet "pac2" can have any source IP address.

Definition (Background Network Field Language)
For any network field, <field_name>, of a legacy firewall policy the binary relation given by, field_name(Packet,Value), is the corresponding network field (binary) relation where Packet is a packet identifier and Value ranges over the possible values of the given network field. ◊

Using these network field binary relations we can capture the network conditions of the firewall rules in a simple auxiliary theory specifying when a network field takes its various values. Then given such a background auxiliary theory of network conditions we can translate (up-lift) any legacy firewall policy to a preference policy in LPP as follows.

Definition (Corresponding Argumentation Theory)
Let FP be a legacy firewall policy. Then the corresponding logical preference policy in Logic Programming with Priorities, denoted by L(FP), is given as follows. For every filtering rule:

```
<order_num> : <action_value>  if
                    <protocol_value><src_ip_value><src_port_value>
                    <dst_ip_value> <dst_port_value>
```

we have the corresponding rule in the basic part of L(FP) given by:

```
r(order_num) : action(Packet, action_value) if
                    protocol(Packet,protocol_value),
                    scr_ip(Packet, src_ip_value),
                    scr_port(Packet, src_port_value),
                    dst_ip(Packet, dst_ip_value),
                    dst_port(Packet, dst_port_value).
```

The priority or strategy part of L(FP) is given by the single priority rule:

```
R(total): priority(r(Name1), r(Name2)) if Name1 < Name2.
```
◊

This is a straightforward translation expressing in logical terms each filtering rule and the total ordering amongst these rules by giving higher priority to any rule over all the rules that are named with a higher number. For example, if our firewall policy has the default rule to deny any traffic on some protocol, ptc1, then the corresponding rule in the basic part of *L(FP)* will be:

```
r(default_num) : action(Packet, deny) if
                    protocol(Packet,ptc1),
                    scr_ip(Packet, any),
                    scr_port(Packet, any),
                    dst_ip(Packet, any),
                    dst_port(Packet, any).
```

Similarly, borrowing the example policy from [37] the following row in this example

```
2 : tcp, 140.192.37.2*, any, any, 80, accept
```

will be translated to the rule:

```
r(2) : action(Packet, accept) if
                    protocol(Packet,tcp),
                    scr_ip(Packet, net_140_192_37),
                    scr_port(Packet, any),
                    dst_ip(Packet, any),
                    dst_port(Packet, 80).
```

The following theorem shows the inclusion of legacy firewall policies into the framework of Logic Programming with Priorities and in particular into the framework of LPwNF.

Theorem (Inclusion of Legacy Policies)

Let FP be a legacy firewall policy and L(FP) its corresponding logical preference policy in LPP. Then for any given packet of traffic, Packet:

$$\text{FP}(\text{« Packet: Action »}) \Leftrightarrow \text{L(FP)} \models_{spr} \text{action(Packet, Action)}.$$

This theorem therefore tell us that the decision of a firewall policy, to accept or deny any packet of traffic, is exactly mirrored by the argumentation based preference reasoning in the corresponding logical policy. The operational semantics of a legacy firewall policies is now captured declaratively through the logical semantics of argumentation based preference reasoning.

A.4.2 Extending Legacy Firewall Policies

The inclusion we get by the above theorem of legacy firewall policies into LPP is a proper inclusion as we can express policies in LPP that cannot be captured by a legacy policy. Indeed, the form of the corresponding preference policies *L(FP)* in LPP is a very special one in several ways such as:

1. Background Language for Network Conditions
2. Action Naming Field
3. Ordering of the Policy Rules
4. Decomposition of Policy into Components

The high-level of expressivity afforded by the framework of LPP opens up the possibility to consider several types of extension of legacy firewall policies along all the lines indicated above. Here we will indicate various different possibilities of extension of legacy firewall policies through illustrative examples. Of course, any extension considered should have its practical motivation.

It is important to stress that the main focus of all such extensions is to allow us to represent network security policies directly from their high-level specification of security requirements. It is a first step towards using the framework of LPP as a tool for the specification of requirements of network security software (and, as we will see in the next section, for the analysis of such requirements).

Consider for example a network security policy:

"Deny traffic from outside, i.e. where the source IP belongs to a network that is different from the network of the destination IP, unless this comes from a secure source".

In order to capture this policy in a direct way we can define in our background logical theory the auxiliary relations "outside_traffic(Packet)" and "secure_source(Packet)" as follows:

```
outside_traffic(Packet) if
                src_ip(Packet, Net1),
                dst_ip(Packet,Net2),
                Net1 ≠ Net2.

secure_source(Packet) if
                src_ip(Packet, Source),
                secure_ip(Source).
```

where "secure_ip" is given by a set of facts cataloguing which machines or networks are secure. Then given these auxiliary definitions we can express the network policy directly through the following policy rules:

```
r(k) : action(Packet, accept) if
                     outside_traffic(Packet),
                     secure_source(Packet).

r(k+1) : action(Packet, deny) if
                     outside_traffic(Packet).
```

and the priority that r(k) has over r(k+1).

This network policy statement is very general and could span over several other policies that we want to apply, i.e. that irrespective what the specific policies say we want that outside traffic (inter network traffic) to be blocked unless this comes from a secure source. In such a case this would mean that in effect we want this policy to be a meta-policy that would have effect over other specific policies. We should therefore be able to state a "global" priority of this policy over the other policies. In fact, this is a matter of distributed policies and how we compose them together. We will address this issue below where we will discuss extending the naming field of the policy rules to refer to the policy to which each rule belongs.

Let us though indicate further the problem by considering that we are given in addition another security requirement as follows:

"Deny traffic from network, net_1, to network, net_2, unless the destination IP is the machine, aias."

In a similar way as above this is easily captured by the following two rules:

```
r(m) : action(Packet, accept) if
                     scr_ip(Packet, net_1),
                     dst_ip(Packet, aias).

r(m+1) : action(Packet, deny) if
                     scr_ip(Packet, net_1),
                     dst_ip(Packet, net_2).
```

and the priority of r(m) over r(m+1).

The problem now is the relative ordering between these two sets of two rules, i.e. should this latter set of rules be above or below the rules for the policy requirement for outside traffic given above? One may be tempted to say that since the latter requirement appears to refer to cases that are more specific that they should go above the previous rules, e.g. as:

```
r(n) : action(Packet, accept) if
                     scr_ip(Packet, net_1),
                     dst_ip(Packet, aias).

r(n+1) : action(Packet, deny) if
                     scr_ip(Packet, net_1),
                     dst_ip(Packet, net_2).

r(n+2) : action(Packet, accept) if
                     outside_traffic(Packet),
                     secure_source(Packet).

r(n+3) : action(Packet, deny) if
                     outside_traffic(Packet).
```

thus giving a total order amongst these rules with r(n) the strongest and r(n+3) the weakest.

But this would then have the consequence of denying outside traffic from net_1 to net_2 that was coming from a secure source within net_1 (except for the special case where this has destination aias). Hence the first requirement would not be correctly represented. The problem stems from the fact that in legacy firewall policies the two rules "r(n+1)" and "r(n+2)" need to be ordered and this has the unwanted side-effect of "r(n+1)" preventing the operation of "r(n+2)". Similarly, if we used the opposite ordering we would deny traffic from net_1 to aias (in net_2) whenever the source (in net_1) of the traffic is not secure. Finally, if we tried to interleave the two sets of rules, e.g. as:

```
r(n) : action(Packet, accept) if
                    scr_ip(Packet, net_1),
                    dst_ip(Packet, aias).

r(n+1) : action(Packet, accept) if
                    outside_traffic(Packet),
                    secure_source(Packet).

r(n+2) : action(Packet, deny) if
                    scr_ip(Packet, net_1),
                    dst_ip(Packet, net_2).

r(n+3) : action(Packet, deny) if
                    outside_traffic(Packet).
```

we would get the unwanted behaviour of accepting traffic from net_1 to aias even when this is not coming from a secure source in net_1 (going against the secure source policy requirement).

The above example illustrates the limitation of existing framework for firewall policies where policy rules need to be totally ordered. This need of totally ordered rules forces the different rules to interact with each other in an ad hoc way (as we have seen a different choice for the total order gives a different interaction) when this is not meant by the policy maker. In our proposed framework of LPP the ordering amongst the rules is allowed to be partial and this gives us the flexibility to capture more complex security requirements. In the above example, we would have in the strategic part of the LPP theory, *TP*, encoding the policy the rules:

```
R(k): r(k) > r(k+1)
R(m): r(m) > r(m+1).
```

Note that here and below we will present priority rules using the infix notation ">" instead of the priority/2 predicate.

Then the decision of which of these two component policies is stronger when they are both relevant, e.g. in the case of traffic from net_1 to net_2 which comes from a secure source, becomes an explicit part of the policy and is represented by an appropriate rule, e.g.:

```
R(1): r(k) > r(m+1) if secure_source(Packet)
```

expressing the fact that traffic from a secure source should be accepted. In effect, we are stating that the secure source policy is stronger than the policy for regulating traffic from net_1 to net_2.

Note here the ***conditional*** nature of the ordering of the policy rules. We can use this to provide further flexibility in our policy representation. Consider for example that

```
R(2): r(m+1) > r(k) if dst_ip(Packet,mail_server)
```

What happens though if we have traffic (from net_1 to net_2) which comes from a secure source and is destined for the mail-server? Both these priorities apply and we have a situation where our policy has an argument to "accept" and an argument to "deny": the policy is in a dilemma. To resolve this we need to consider which of these two policy orderings is the strongest: is the secure source policy stronger than the policy regulating traffic from net_1 to net_2 to the mail_server of net_2 or vice versa? Suppose that the latter is true. We can capture this using a ***higher-order*** priority ordering between these two ordering rules:

C: R(2) > R(1)

that would then take us out of the dilemma: only the argument for denying such traffic would be admissible and so the policy would now have a deterministic sceptical conclusion for "deny".

In fact, such higher order priority rules can themselves be conditional so that R(2) is stronger than R(1) in some circumstances and weaker than R(1) in other circumstances. We can then repeat the process of detecting dilemmas in the decision of our policy and introduce a rule at a new higher level of priority to specify which of these statements is stronger when they both apply. So for example, if we have

C(2): R(2) > R(1) if peak_time_net_2
C(1): R(1) > R(2) if source_ip(Packet, kronos)

we would employ, according to the requirement that is very important for traffic from kronos to reach its destination as soon as possible even at the peak time for net_2, a rule at one level higher in the priority ordering such as:

D: C(1) > C(2)

to capture this requirement.

Appealing to higher levels of ordering can in theory continue at infinitum. In practice, when we are developing security policies we can follow a systematic methodology to detect such dilemma points and resolve with the help of the user/developer by referring to a higher level decision of which component policy is stronger or under which conditions one policy is stronger and weaker. In this way we build a natural hierarchical ordering structure amongst the lower-level policy rules that generate the recommendation and the rules that specify the ordering amongst these and the ordering rules themselves.

A.4.3 Composing Policies

A natural way to facilitate this process of the development of security policies is to enhance the language of representation of policy rules so as to refer explicitly to the policy they belong to and then give overall (perhaps conditional) priorities amongst groups of rules. In our approach we can do this by extending the ***naming field*** of the rules (generation or priority) as allowed naturally by the framework of LPP beyond the simple arithmetic name that is essentially the only name employed by legacy firewall policies.

Definition(Naming of Policy Rules)
The name of a firewall policy rule is a triple:
<id><TrafficName><PolicyName >
where "id" is a unique identifier of the rule, "TrafficName" is a term that names (part of) the network traffic and "PolicyName" is a term that names the (component of the) policy to which the rule belongs. ◊

The specific structure of these naming fields is left open and can be chosen by the developer as best suited for the particular policies that are being developed. For our example above in this section we could chose the TrafficName field to indicate the source network/machine, the destination network/machine and the protocol. Then the rule that we have called r(2) above will now be called

r(<d(2), traffic(net_140_192_37,any,tcp), policy(basic)>)

where we have also named the policy to which these rules belong to as "basic". The identifier, d(2), is used also to record that this rule is a denying rule. Similarly, the rules r(k) and r(k+1) above will be name as

r(<a(k), traffic(any,any,tcp), policy(outside_traffic)>)
r(<d(k), traffic(any,any,tcp), policy(outside_traffic)>)

and the rules r(m) and r(m+1) will have names:

```
r(<a(m), traffic(net_1,aias,tcp), policy(net_1net_2)>)
r(<d(m), traffic(net_1,net_2,tcp), policy(net_1net_2)>)
```

With this extension of the naming field we now have the enhanced flexibility of expressing directly ordering requirements. For example, if we have a requirement that says "always accept traffic that has a certain property, e.g. comes from a secure source" then we would represent this with a priority rule of the form:

```
R(Id, Traffic, Policy) :
r(Id1, Traffic, Policy1) > r(Id2, Traffic, Policy2) if
                                acceptance_rule(Id1), property(Traffic)
```

e.g. when the property is that of secure source this rule will be:

```
R(over(Id1,Id2), Traffic, policy(secure_source)) :
r(Id1, Traffic,Policy1) > r(Id2, Traffic, Policy2) if
                                acceptance_rule(Id1), secure_source(Traffic)
```

Note that this rule covers the rule R(1) we have above, but now this is expressed in a declarative way directly from the specification of the requirement.

Similarly, we can use the Policy name field to express relative strength requirements amongst different components of our policy in a declarative and direct way. For example we could have the requirement that "Any Security Policy is stronger than any Business Policy". We would then capture this simply by the rule:

```
R(over(Id1,Id2), Traffic, policy(security_over_business)) :
r(Id1, Traffic,policy(Policy1)) > r(Id2, Traffic, policy(Policy2)) if
                                        security(Policy1), business(Policy2)
```

This requirement may be too general and we may want to apply it only to some specific policy components or under some additional conditions. For example, we may want that "policy_1 is stronger than policy_2 for traffic coming from net_0. Otherwise, policy_2 is stronger." We would then capture this directly as follows:

```
R(over(Id1,Id2), Traffic, policy(policy_1_over_policy_2)) :
r(Id1, Traffic, policy(policy_1)) > r(Id2, Traffic, policy(policy_2)) if
                                        traffic_source(Traffic, net_0)

R(over(Id1,Id2), Traffic, policy(policy_2_over_policy_1)) :
r(Id1, Traffic, policy(policy_2)) > r(Id2, Traffic, policy(policy_1)) if
                                traffic_source(Traffic, Net), Net ≠ net_0
```

In this case there is no need to go to any higher level of priority ordering as these two rules are exclusive in their conditions: only one can apply in any one case. In other cases this may not be so. For example, consider that the requirement was instead: "policy_1 is stronger than policy_2 for traffic coming from net_0. On the other hand policy_2 is stronger when we have low_bandwidth." Then the second rule will be replaced by

```
R(over(Id1,Id2), Traffic, policy(policy_2_over_policy_1)) :
r(Id1, Traffic, policy(policy_2)) > r(Id2, Traffic, policy(policy_1)) if  low_bandwidth
```

and we would also need a higher-order priority rule to capture the requirement that when both of these apply, i.e. we have traffic coming from net_0 at a time of low_bandwidth, policy_1 is stronger

```
C(over(Id1,Id2), Traffic, policy(policy_1_over_policy_2)) :
        R(over(Id1,Id2), Traffic, policy(policy_1_over_policy_2)) >
                R(over(Id1,Id2), Traffic, policy(policy_2_over_policy_1))
```

It is clear that the extension of the naming field that we have proposed above can be particularly useful when we consider distributed firewall policies and how their component policies should be composed together. This composition depends on the topology of the network and the "meta policy" that we want to apply in putting these together. The naming of our rules is well suited to address this problem. In practice though we would need to tame the expressiveness of our naming language and restrict to a useful subset that is sufficient for the needs of real life practical applications. A systematic process of naming the rules within such a restricted language needs to be developed.

We close this section by discussing briefly some other possibilities of extending legacy firewall policies afforded by the more general framework of LPP into which we have mapped these policies. These are extensions that the LPP framework of argumentation gives for free but we would need to examine further their practical value in the application of network security policies.

In the same way we have extended the network field of our rules we can consider extending the Action Field of our rules. We could for example introduce a third action, called Delay or Suspend so that we can capture policy requirements that instead of denying traffic they simply delay this. For example, we may want to suspend some type of traffic when the available bandwidth is low so that other traffic can move faster. We would then employ rules such as:

```
r(<de(k), traffic(any,any,tcp), policy(delay_outside)>) :
    action(Packet, delay) if outside_traffic(Packet), low_bandwidth.
```

to delay outside traffic at a time of low bandwidth so that Intranet traffic is delivered quickly.

Another extension is to consider policy requirements which link two different policies, i.e. the decision of one policy depends on the decision of the other policy. We could have for example rules of the form:

```
<policy1: action1> if <policy2:action2><network conditions>
```

where under certain conditions the action recommended by policy1 depends on the recommendation of policy2. An example, of such a policy rule could be used to represent the security requirement that

"The department's local firewall should accept any traffic with destination the student sub network if this has been accepted through the main university firewall."

```
r(a(k), traffic(any, net_student, policy(Dept_Policy)) :
action(Dept_Policy, Packet, accept) if action(Uni_Policy, Packet, accept),
                                        dst_ip(Packet, net_student)
```

We can even have more elaborate interactions between policies where the decision of one policy depends on the decisions of several other policies, i.e. employing rules of the form:

```
<policy: action> if <policy1:action1><policy2:action2>…<policyk:actionk>
                    <network conditions>
```

An example of such a rule would be useful when we have a cascade of firewall policies, generalizing the above example of University Policy and Departmental Policy to a sequence of more than two links. Another possibility would be that a policy is composed of several other policies each one taking into account one aspect of the security or other (e.g. economy or quality of traffic) desirable properties of the network operation. For example, we may have a requirement:

"Any traffic that is denied under policy 1 and policy 2 should be accepted under policy 3."

stemming from our network topology that at some point in the network, traffic can be send through to one of three possible routes and destination points and that for each of these routes we have a separate policy to decide if it can be allowed to take this route. The requirement above ensures that in the worst (or default) case traffic will go through the third route as security in this sub network is not paramount.

We can therefore see that such an extension with rules that explicitly represent interdependencies of different policies may be useful in capturing distributed polices coming from different topologies of networks and the associated requirements of composition of their component firewall policies.

Finally, we note that it is also possible that the ordering rules of one policy can be conditional on decisions of other policies, i.e. we can have rules of the form

```
R(over(Id1,Id2), Traffic, policy(Policy1)) :
r(Id1, Traffic, policy(Policy1) > r(Id2, Traffic, policy(Policy1) if
                         action(Policy2, Action), network_conds(Traffic).
```

A.5 Analysis of Firewall Policies

Firewall policies can suffer from the problem of *anomalies*. In its most general term this is the problem where:

"one part of a policy is in some way in conflict with another part of the policy".

In legacy firewall policies this problem is exasperated by the fact that these policies have a very rigid structure where all rules must be in a total order of priority. As the number of rules in the policy grows the problem of existence of anomalies becomes more severe. It is therefore necessary to develop methods to analyse a policy and detect anomalies and then provide a systematic methodology to resolve these anomalies.

Current studies of this problem defined the anomalies through a syntactic analysis of the policy rules and use specialised algorithms to detect such anomalies. Different notions of the possible relations between two rules are defined by a purely syntactic comparison of the conditions of these rules under which a conflict between the rules can arise. In other words, the general and abstract semantic definition of anomaly is reduced to a set of syntactic relations amongst the rules which, although in some cases (e.g. isolated firewall policy) can be argued to be complete, as we extend our policy framework, e.g. to include distributed firewall policies, this approach of projecting the general and intuitive notion of anomaly and conflict onto a set of *single* rule relations becomes ad hoc and incomplete.

In this section we will show how we can analyse firewall policies using the semantic notions of preference reasoning through argumentation capitalizing on the declarative representation and semantics for firewall policies that we have developed in the previous section of modelling network security policies. Anomalies will thus be captured moving away from the procedural interpretation of the policies through semantic definitions that *remain invariant* as we develop further the type of our policies, e.g. as we consider extensions of policies such as distributed policies.

We will start by showing how the various types of anomalies, as given in [37], for single legacy firewall policies can be captured naturally using the basic notions of argumentation once these policies are mapped into theories of LPP.

- **Shadowing Anomaly:** A rule, r(n), is never activated (because a previous rule matches all its packet cases). The rule therefore appears to be redundant.

Definition (Shadowing Anomaly 1)
Let FP be a legacy firewall policy. Then FP contains a shadowing anomaly iff there exists two rules, r(n) and r(k), in the corresponding logical preference policy L(FP) such that, for a particular network traffic, there exists two admissible arguments one containing r(n) and the other r(k) when we have deleted from L(FP) the relative ordering between these two rules . ◊

We can easily see that this is then equivalent to the following definition that refers to the policy as given without any alterations.

Definition (Shadowing Anomaly 2)
Let FP be a legacy firewall policy. Then FP contains a shadowing anomaly if and only if there exists a rule, r(n) in the corresponding logical preference policy L(FP) such that there is no admissible argument containing r(n). ◊

Note that these definitions require that we consider the conclusions of any two rules in the policy to be incompatible with each other. This may be considered as unnatural. We will later see that we can avoid it, if we wish, when we give a general definition of anomaly that encompass all different existing notions of anomalies into a single notion, where some types of anomalies simply do not appear anymore. We are using this incompatibility amongst all rules here to capture exactly the existing literature.

- **Correlation Anomaly:** The relative order of two rules is significant: two rules have an overlap of packets that apply to them both.

Definition (Correlation Anomaly)
Let FP be a legacy firewall policy. Then FP contains a correlation anomaly if and only there exists two rules, r(n) and r(k), in the corresponding logical preference policy L(FP) such that for a particular network traffic, there exists two admissible arguments one containing r(n) and the other r(k) when we have deleted from L(FP) the relative ordering between these two rules . ◊

An equivalent way to capture this anomaly is to have that in L(FP) there is no admissible argument containing a rule r(k) whereas when we remove a rule, r(n) (n<k), from L(FP) (or when we remove the ordering, r(n) > r(k), from L(FP)) then there is an admissible argument containing r(k).

- **Generalization Anomaly:** This is the dual of the Shadowing anomaly. An exception rule appears before a general rule (as it should if it is to act as an exception).

Definition (Generalization Anomaly 1)
Let FP be a legacy firewall policy. Then FP contains a generalization anomaly if and only if there exists two rules, r(n) and r(k) (n <k), in the corresponding logical preference policy L(FP) such that there exists two admissible arguments one containing r(n) and the other containing r(k) when the relative ordering of r(n) > r(k) in L(FP) is removed. ◊

Equivalently we have the following formulation.

Definition (Generalization Anomaly 2)
Let FP be a legacy firewall policy. Then FP contains a generalization anomaly if and only if there exists two rules, r(n) and r(k) (n <k), in the corresponding logical preference policy L(FP) such that there exists no admissible argument containing r(n) when the relative ordering of r(n) > r(k) in L(FP) is reversed. ◊

Finally, let us consider the redundancy anomaly where a rule in the policy never applies.

- **Redundancy Anomaly:** A general rule has higher priority than a rule (with the same action field) which is a specific case of the general rule. The specific rule is therefore redundant as it will never apply.

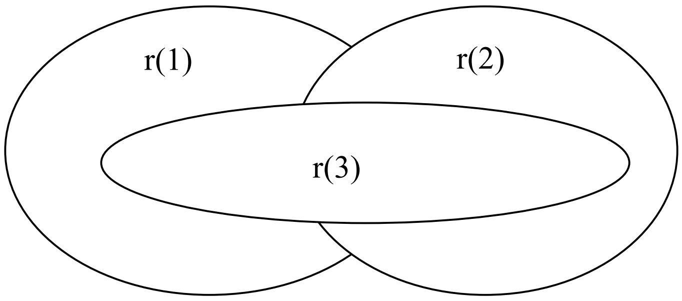
Definition (Redundancy Anomaly 1)
Let FP be a legacy firewall policy. Then FP contains a redundancy anomaly if and only if there exists two rules, r(n) and r(k) (n <k), in the corresponding logical preference policy L(FP) whose action field is the same such there exists two admissible arguments one containing r(n) and the other r(k) when we have deleted from L(FP) the relative ordering between these two rules and we consider the conclusions of these two rules as incompatible. ◊

Again in the given policy with out any modifications this can be captured equivalently as follows.

Definition (Redundancy Anomaly 2)
Let FP be a legacy firewall policy. Then FP contains a redundancy anomaly if and only if there exists two rules, r(n) and r(k) (n <k), in the corresponding logical preference policy L(FP) whose action field is the same such that there exists no admissible argument containing r(k) when we consider the conclusions of these two rules as incompatible. ◊

In other words, if we consider an incompatibility notion that makes only these two rules incompatible then one of the rules becomes redundant in the sense that we will never have an admissible argument for it. Note that this anomaly is the same as the Shadowing Anomaly but where now the conclusions of the rules have the same action.

The above equivalent semantic reformulation of the anomalies can help us reveal a uniform understanding of the notion of anomaly in these policies. On the one hand it helps see the ad hoc nature of the decision to select these anomalies over others. For example, we see that essentially the two types of Shadowing and Redundancy anomalies are the same. Also why should we restrict ourselves to considering only single rules in pairs. We could instead consider subsets of rules. We could have a shadowing anomaly where two rules shadow another rule in a way that this can not be reduced to shadowing between the single rules. This can happen when the network conditions of the rules r(1), r(2) and r(3) have the following set relationship:



Then {r(1),r(2)} together shadow {r(3)} but there is no shadowing between {r(1)} and {r(3)} or between {r(2)} and {r(3)}.

The semantic reformulation of the anomalies shows us that their essential aspect is that of the existence of different admissible subsets, S1 and S2, of our policy that have incompatible conclusions. In the extreme case we have that there is no admissible subset S1 (containing a certain rule and its conclusion) but only admissible subsets S2 with incompatible conclusions. In fact, we can see that the correlation anomaly captures this central notion with the shadowing and redundancy anomalies being extreme cases.

This leads us to the following general definition of anomaly within the logical framework of LPP where we now represent our network security policy as an argumentation-based preference policy as described in the previous sections.

Definition(Anomaly in LPP Firewall Policies)
Let P be a theory in LPP representing a security policy. Then P contains an anomaly if and only if

1. **[conflict anomaly]** either there exists a particular network traffic and two subsets, P1 and P2 of P, such that P1 and P2 are admissible arguments and have incompatible conclusions,

2. **[redundancy or extreme conflict anomaly]** or there exists a subset P1 of the basic part of P such that P1 is not admissible for any particular network traffic that satisfies the conditions of P1. ◊

As we have shown above it is easy to prove the following result.

Theorem (Inclusion of Anomalies)
Let FP be a legacy firewall policy and L(FP) its corresponding logical preference policy. If FP contains any one of the four anomalies, shadowing, correlation, generalization and redundancy, then L(FP) also contains an anomaly.

We note that the above definition of anomaly in firewall policies, written as theories of LPP, corresponds to what are normally considered as "standard problems", or lack off desired properties (see section A.1), in any argumentation based preference theory. These are on the one hand, the existence of non-determinism and hence of a dilemma in deciding what is the (preferred) conclusion of the theory and on the other hand the existence of a part of the theory, i.e. of some arguments, that will never be applied and hence this part is apparently redundant in the theory.

A.6 Detecting Anomalies – Link with Abduction

Given the above universal notion of anomaly the task of detecting anomalies in a given policy reduces to the simple task of posing a set of semantic queries on the policy. No specialised algorithms are needed. Instead we can use the basic semantic query of an argumentation-based framework:

```
arg-prove(Goal,Delta)
```

of showing the existence (and constructing) of the subset, Delta, of the given policy as an admissible argument for the goal, Goal.

In our case the Goal is of the general form, action(NetTraffic,Action), and based on this we have the following general query schema for detecting anomalies:

```
anomalous1(NetTraffic,Action) if
   not arg-prove(action(NetTraffic,Action),Delta)

anomalous2(NetTraffic) if
        arg-prove(action(NetTraffic,Action1), Delta1),
        arg-prove(action(NetTraffic,Action2), Delta2),
        incompatible(Action1,Action2).
```

Given a specify network traffic of interest, net_traffic, and a specific action, act_1, of interest for this traffic the queries:

```
? anomalous1(net_traffic, act_1)

? anomalous2(net_traffic)
```

show, if anyone of the two succeeds, the existence of an anomaly for this specific network traffic. With the first query we detect that for this type of traffic we can never have the action, act_1, revealing a redundancy type of anomaly (or as explained above an extreme conflict anomaly). With the second query we detect that for this type of traffic we have at least two possible but incompatible actions, revealing a conflict anomaly. If both these queries fail then the policy is anomaly free.

Note that the second query when it succeeds gives us also the arguments Delta1 and Delta2 that support the conflicting conclusions of the policy. This facilitates the process of resolution of the anomaly as we can see in these which subparts of the policy are in conflict (see below the subsection on anomaly resolution).

The above queries assumed that we first select a specific network traffic that we suspect might be treated in an anomalous way by the policy or we in some way generated systematically all the different possible traffics within our network and we check each one for an anomaly. In order to find out automatically such cases of anomalous traffic we can employ, together with argumentation, the reasoning process of abduction that would generate (if they indeed exist) the specific traffic or scenarios for which the policy has an anomaly. This application of abduction is completely analogous to its use in the earlier studies of the problem of policy analysis and refinement in [44] and later in the context of [1].

Hence we would employ the semantic query:

```
abd-arg-prove(Goal,Delta,Hypotheses)
```

which generates a set of abductive hypotheses, Hypotheses, on a a-priori chosen set of relations (in our case the relations describing the network traffic) which when we take as given extend our policy such that the goal, Goal, has the admissible argument, Delta. In other words, these hypotheses describe a specific scenario under which the Goal is admissible. In our case they will describe a specific case of network traffic, in terms of the network conditions which will be our abducible predicates, for which the Goal is admissible.

Our queries for finding anomalies will then be based on extended definitions these. For example, the definition of "anomalous2" given above will be extended to:

```
anomalous2(NetTraffic) if
  abd-arg-prove(action(NetTraffic,Action1), Delta1,Hyp),
  abd-arg-prove(action(NetTraffic,Action2), Delta2,Hyp),
  incompatible(Action1,Action2).
```

showing that under the common hypotheses (scenario), Hyp, that specify the network traffic, NetTraffic, we have admissible arguments for two incompatible actions for this traffic. A query

```
?anomalous2(NetTraffic)
```

that leaves the network traffic open will then generate scenarios, described by the hypotheses Hyp, under which the policy has an anomaly.

We mention here that abduction can be used in order to address other problems than the detection of anomalous scenario (traffic). An important such problem would be the task of finding under which priority ordering relations of the rules of our policy we could ensure that some desired property held true. Suppose for example that we want the safety requirement that any traffic that comes from a Domain Name Server (DNS) is to be accepted. How can we ensure this?

We could at first check, using the query

```
? arg-prove(action(dns_traffic,accept), Delta),
  not arg-prove(action(dns_traffic,deny), Delta1)
```

to check that there is indeed at least one admissible argument to accept this and that there is no admissible argument, Delta1, in our policy that would deny such traffic. If this query succeeds then the safety requirement holds and there is nothing else to do. But if it fails we will need to edit our policy so that this cannot happen. In particular, as we mentioned above, we may want to find under what ordering of the policy rules this can succeed. For this we then declare our abducible relation to be the ordering relation amongst rules. We could for example introduce a general abducible relation, abd_order(Id1,Id2), which using the general schema rule

```
r(Id1) > r(Id2) if  abd_order(Id1,Id2)
```

would allow us to generate hypotheses on the ordering of the rules of our policy through ground hypothetical facts of abd_order(Id1,Id2). Then by posing the abductive query

```
? abd-arg-prove(action(dns_traffic,accept), Delta, Hyp),
not abd-arg-prove(action(dns_traffic,deny), Delta1,Hyp)
```

we would generate a set of ordering hypotheses, Hyp, that would ensure that this traffic can only be accepted as required.

A.7 Anomalies in Distributed Policies

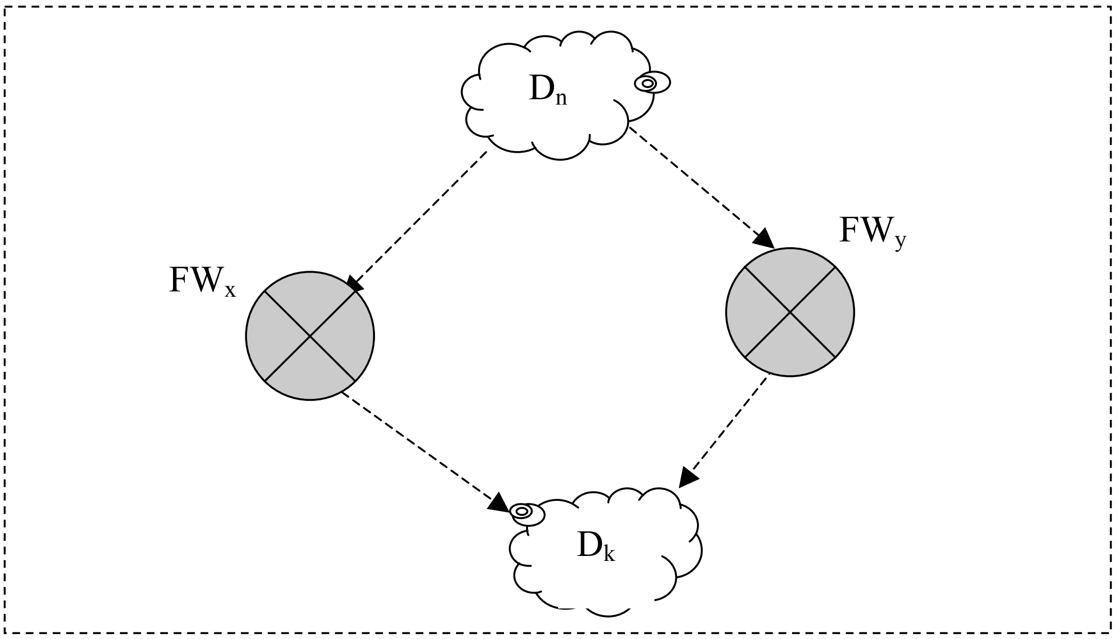
With the semantic definition of anomaly defined above, we can now escaped from the procedural (and single rule) realm of anomalies and we can easily use (or generalize) this same notion to distributed policies, where we will be looking for components of the distributed policy that are ***admissibly incompatible***. This has become possible because even in the case of a single policy we now view anomalies as cases of conflict between its different parts, i.e. as a policy distributed amongst its parts.

With distributed policies the problem of the existence of anomalies is more severe as there are more possibilities for conflict to occur. In fact, in a distributed policy we can have situations where one component decides to accept a traffic whereas another component decides to deny it. This can not occur in a single legacy firewall policy due to the linear and total order of its parts.

For example, we can have an upstream firewall blocking a traffic that is permitted by a downstream firewall, a type of inter-firewall shadowing anomaly. In a "classical" operational approach for anomalies the definition of this anomaly requires a detailed (and somewhat ad hoc) examination of the pairs of rules form the two firewalls. In our declarative approach this anomaly falls under the same definition given above. Simply, we have for the same traffic an admissible subset of the upstream policy that concludes to accept it and at the same time we have an admissible subset of the downstream policy that concludes to deny it.

Our general definition of anomaly is indeed all encompassing, able to capture all other types of anomalies of distributed policies. In every case we simply have admissible subsets of two different components of the policy that have an opposite or incompatible conclusion. As we change the network topology we do not need to change this definition and hence our methods of detection of anomalies. We can for any network topology base our investigation of anomalies on the same definition of the existence of admissible components that are incompatible.

For example, instead of a linear network topology where we have cascading firewalls we may have a topology that allows alternative routes to some domains through different firewalls. In this case, the anomalous situation where the same traffic between two domains (e.g. D_n and D_k in the figure below) would be accepted via one route and denied through another route is again simply a case where in the distributed firewall policies involved in these two routes there are two admissible parts that have the opposite conclusion for this traffic. Hence in the figure below, there is a firewall, FW_x within one route and a firewall FW_y within the other route, such that for some traffic the policy FW_x admissibly accepts this whereas the policy FW_Y admissibly denies this. This universality of the definition of anomaly which simply formalizes the informal notion of "the existence of conflict of opinion between different parts of the policy" is not exhibited by the "classical" operational approach where one needs to change and customize its definition of anomaly to each different particular network topology.



A.8 Anomaly Resolution

The declarative understanding of anomalies and its universality facilitates the problem of their resolution. We can address this problem within the general process of the development of the security policy following the general methodology that we have proposed in section A of this report for the development of specifications of software requirements within the argumentation framework of LPP.

Hence given the existence of an anomaly we would have two parts of the policy that are in conflict and we could use our conflict resolution methods to remove this. Each one of these admissible parts of the policy would contain not only the basic rules, that fire under the traffic conditions for which the anomaly exists, but also the rules that give the priority ordering that makes them admissible. The administrator/developer would then be alerted either (a) to the fact that some ordering relation is mistaken or (b) that the existing ordering rules are incomplete at some level or (c) new ordering rules are needed at one level higher to address the question of which rules are stronger.

In particular, in the case of distributed policies the administrator would be alerted to the fact that two component policies are in conflict and would then be able to resolve this using a meta-policy of priority amongst the two components. For example, in order to resolve an inter-firewall shadowing anomaly, the administrator can apply the meta-policy that the more local (downstream) a firewall is in a cascade of firewall policies the stronger it is, by employing the priority rule:

```
R(over(Id1,Id2), Traffic, policy(downstream_stronger)):
r(Id1, Traffic, policy(Policy1))  > r(Id2, Traffic, policy(Policy2))  if
                                                    downstream(Policy1, Policy2)
```

Another way to represent this meta-policy within LPP would be to state directly the priority on the conclusions of the policies as follows:

```
R(over(Policy1,Policy2), Traffic, policy(downstream_stronger)):
action(Policy1, Traffic, Action1)  > action(Policy2, Traffic, Action2)  if
                                                    downstream(Policy1, Policy2)
```

More generally, this meta-policy of stronger downstream firewall may apply only under some specific circumstances of the traffic for which we have the anomaly and that in the other circumstances the upstream firewall should be stronger. Then these meta-policy rules would be further conditioned on these circumstances.

A.9. Modelling & Analysing Policies in GORGIAS

The LPP framework of LPwNF is implemented in a system called GORGIAS [45]. This supports both the argumentation reasoning of LPwNF and its abductive reasoning. It is build on top of Prolog and it is available at: www.cs.ucy.ac.cy/~gorgias. Details of how to use this system with tutorial examples can be found at this web page. This system has been used recently in the development of an agent architecture [46] to provide the agent with a flexible goal decision mechanism and at the same time a declaratively specified operational model of the agent that is adaptable to its external environment. It has also been used in the development of applications of automated decision making in the medical domain [47].

In appendix A the full implementation in GORGIAS of the Elevator Policy presented in section A is given, together with some examples scenarios that illustrate its use in the operation of the elevator. Implementing legacy firewall policies in GORGIAS can be done in the same way and in fact this is simpler due to the simpler nature of these policies. Here are two example rules in the basic part of an example policy (taken from [45]):

```
% Deny all packets from machine "apollo" to port 80
rule(basic(1, [tcp,apollo,SrcPort,DstIp,80]), action(deny, Packet),
                    [sourceip(Packet,apollo), destinationport(Packet,80)]).

% Accept all packets from network 1 to port 80.
rule(basic(2, [tcp,net1,SrcPort,DstIp,80]), action(accept, Packet),
                    [sourceip(Packet,net1), destinationport(Packet,80)]).
```

Here the predicates sourceip/2, destinationport/2 etc are auxiliary background predicates defined by simple rules, e.g.

```
sourceip(packet(Protocol,ScrIp,SrcPort,DstIp,DstPort),net1):-
                    ScrIp = [140,192,37,X].

sourceip(packet(Protocol,ScrIp,SrcPort,DstIp,DstPort),apollo):-
                    ScrIp = [140,192,37,20].

destinationport(packet(Protocol,ScrIp,SrcPort,DstIp,DstPort),DstPort).

destinationip(packet(Protocol,ScrIp,SrcPort,DstIp,DstPort),zeus):-
                    DestIp = [160,120,33,40].
```

Note how the name, e.g. "basic(1, [tcp,apollo,SrcPort,DstIp,80])", of the rules in the basic part of the policy includes a representation of the traffic for which they apply. The names do not include a representation of the policy to which they belong as we are representing here only one policy.

The strategy part of the policy as shown by the results of the translation of legacy firewall policies in LPwNF consists simply of the rule:

```
% Priority of rule N1 over rule N2
rule(over(N1,N2),prefer(basic(N1,Packet),basic(N2,Packet)),[N1>N2]).
```

The main query predicate of GORGIAS is:

```
        ?prove(Goal,Argument)
```

which succeeds by finding an admissible argument, Argument, as a list of policy rule names under which the goal, Goal, is logically entailed by these rules.

Hence to find out the recommended action for an example packet of traffic we use queries on "find_action/3" defined as:

```
find_action(Example, ReccomendedAction, SupportingArguments):-
        examplePacket(Example, ExamplePacket),
        prove([action(ReccomendedAction, ExamplePacket)],
                                                SupportingArguments).
```

Automatic detection of anomalies in policies can be carried out using semantic queries on predicates such as anomalous/3 defined by:

```
anomalous(Traffic, Delta1, Delta2):-
        examplePacket(Traffic, ExamplePacket),
        prove([action(ReccomendedAction, ExamplePacket)], Delta1),
        complement(action(ReccomendedAction, Packet),
                   action(CompAction, Packet)),
        prove([action(CompAction, ExamplePacket)], Delta2).
```

Finally, we comment that once a firewall policy is analysed through its GORGIAS implementation we can examine ways how to automatically translate these policies into lower level specialised procedural languages that could effectively employed on real time traffic.

A.10 Discussion

We have studied the use of argumentation based preference policies as a way to specify and analyse software requirements policies. We have established this new link between software requirements policies and argumentation and have shown important properties of this approach:

- Declarative modelling of policies where these can now be represented directly from the requirements specification exploiting the high-level of expressivity of the adopted logical framework of argumentation.
- Semantic high-level analysis of policies with a uniform formulation of anomalies that is invariant as we extend the type of policies that our applications require.
- Automatic detection of anomalies through semantic queries of integrated abduction and argumentation.
- Modularity of representation and semantic analysis of the policies that facilitates their incremental development and the composition of distributed policies.

In the particular area of network firewall security policies our work gives a declarative meaning to these policies and offers a semantic framework for their analysis, that escapes from the low level procedural and syntactic analysis of policies that is present in many of the current approaches to firewall policies. Unlike these approaches, our formalization does not need to be reconsidered and the analysis process does not need to be redefined every time we wish to extend the realm of application of the firewall policies.

In the study of the analysis of firewall policies we have shown specifically that the various types of anomalies in firewall policies, identified separately in the literature, can be captured naturally under the same and unified definition based on the standard notion of an admissible argument in Logic Programming with Priorities (LPP). This high level definition means (a) that we are more complete in capturing the notion of anomaly and (b) that our definitions remain invariant as we develop further the type of our policies, e.g. as we consider extensions of policies such as distributed policies.

The high-level of expressivity of the LPP framework, particularly its ability to represent explicitly orderings which can be conditional on some background properties, facilitates greatly the process of conflict resolution within our policies. This can be carried out by modular and incremental extensions of the policy guided by the user by presenting to her/him the conflict scenarios and the parts of the policy (arguments) that lead to the problem.

Part B: Abductive Constraint Logic Programming for Policy Refinement

B.1. Introduction

Effective management of distributed computing resources in an environment with changing requirements, having multiple administrative domains and heterogeneous technology is a key challenge for today's system administrators. Policy-based approaches to systems management are gaining widespread interest because they provide the flexibility that result from separating the rules that govern the behavioral choices in a system from the underlying functionality. This allows systems to adapt their behavior according to the user's requirements and the current environment.

To date, the majority of the research into policy-based management has focussed on languages for specifying policy rules and architectures for deploying and enforcing policies. As a result, a number of different types of policies have been identified. Broadly these can be categorised as management/behavioral policies, which prescribe how a system should respond in a given situation; and security authorisation policies, which define the conditions under which users or agents can (or cannot) access particular resources. When writing policies it is important to ensure that they meet the user's requirement and are consistent with respect to the overall system in which they are to be used. This latter task is referred to as policy analysis and our prior work has shown how an Event Calculus based formalisation of policy-managed systems can provide administrators with the capability to detect a range of inconsistencies [1]. The requirement of translating the high-level requirements of a user into policies that can be enforced by the system was identified as early as 1993 by Moffett and Sloman [2]. They described the successive refinement of high-level goals in a policy hierarchy, called Policy Refinement, and presented the following objectives for a policy refinement process:

- Determine the resources that are required to satisfy the needs of the policy.
- Translate high-level policies into operational policies that can be enforced by the system.
- Allow analysis to verify that the set of lower level policies actually meet the requirements of the high-level policy.

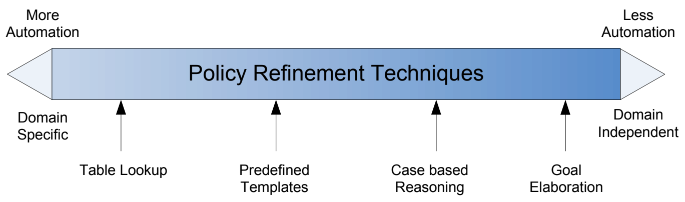


Figure 1: Relationship between policy refinement techniques

Since then, a number of researchers have presented techniques for addressing the challenges of refinement but in each case a number of limitations had to be imposed in order to develop a workable solution. When developing a policy refinement technique the main compromise is between the generality of the technique and the amount of automation. As illustrated in Figure 1, this trade-off means that techniques that are highly automated tend to have a very narrow domain of applications in which they work. On the other hand, techniques that have general applicability are not particularly automated.

Our previous work has shown how a formal representation of policy-managed systems can be combined with the KAOS goal elaboration technique [3], to provide a framework for policy refinement [4]. However, the low-level goals derived using this technique cannot be directly used in policies without first identifying the management 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. We use the KAOS goal elaboration technique 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.

Whilst the use of policy refinement patterns offers a degree of automation in the refinement process, the need to choose subjects, targets and also specify the parameters to be used in the management operations mean that in order to produce an enforceable policy some user input is still required. Additionally, it is necessary to ensure that the policies generated in this manner are consistent with respect to the system, and do not conflict with other policies in the system. In this paper we describe how the formalism used to describe the policy-managed system can be extended such that the abductive inference procedure not only generates potential strategies for achieving a given goal, but also derives the remaining information required to specify a correct policy. We achieve this by using the integrity constraint definition features of the abductive inference procedure. Integrity constraints are rules that specify the conditions under which the formal model of the system being analysis is inconsistent. The main advantage of using integrity constraints is that they can be specified in a modular fashion, making them easier to specify, and are evaluated in a computationally efficient manner.

Integrity constraints with respect to allowable subjects, targets, allowable parameter values and overall system consistency can ensure that the policies derived from the refinement process are correct and complete. However, this process can still result in multiple derived policies that satisfy a given goal. In order to fully automate the refinement process, it would be desirable to have a mechanism to select just one policy from the derived set. We propose integrating a selection algorithm into the automated policy refinement procedure that uses utility functions to quantify the appropriateness of the derived policies and pick the best one.

The remainder of this paper is organised as follows. Section 2 presents some background information about event calculus and abductive reasoning, together with a description of the goal-based policy refinement technique we have already developed. This is followed by a description of the formal language used to represent policy-managed systems in section 3. In section 4 we show how integrity constraints can used to automate the policy refinement procedure and in the following section we provide a discussion of our work. Section 6 describes some related work in the area of autonomic computing and policy refinement, and finally section 7 presents our conclusions and future work.

B.2. Background

B.2.1. Event Calculus and Abductive Reasoning

Event Calculus was first presented by Kowalski and Sergot [5] as a logic-based formalism for representing and reasoning about dynamic systems. Because the language includes a representation of time that is explicit and independent of any (sequence of) events or actions, it has been found suitable for modeling a wide range of event driven systems. This includes systems which are non-deterministic, since it is possible to represent scenarios where multiple events occur simultaneously. Additionally, the implementation does not place any restrictions on the size of the state space of the system being modeled.

Fundamentally, a policy managed system is one whose state changes over time and in response to the particular events. Therefore, the formal representation of such systems must be based on a language that supports the notions of events, time and system states that vary over time. For this reason we chose the Event Calculus as the underlying formal notation for representing policy-based management systems.

Whilst several variations of the Event Calculus have been presented, here we make use of a sorted classical logic form [6], consisting of 3 main sorts: (i) time points that can be mapped to the non-negative integers; (ii) properties that can vary over the lifetime of the system, which we refer to as fluents; and (iii) events. In addition the language includes a number of base predicates, initiates(A, B, T), terminates(A, B, T), holdsAt(B, T), happens(A, T), which are used to define some auxiliary predicates. These are summarised in Figure 2.

- **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 for defining static rules (state constraints).
initiallyTrue(B)	fluent B is initially true.
initiallyFalse(B)	fluent B is initially false.

- **Auxillary predicates:**

clipped(T1,B,T2)	fluent B is terminated between timepoint T1 and T2.
declipped(T1,B,T2)	fluent B is initiated between timepoint T1 and T2.

- **Domain independent axioms:**

clipped(T1, B, T2)	← terminates(A,B,T) ∧ happens(A,T) ∧ T1 [T < T2
declipped(T1, B, T2)	← initiates(A,B,T) ∧ happens(A,T) ∧ T1 [T < T2
holdsAt(B, T1)	← initiallyTrue(B) ∧ ¬ clipped(0, B, T1).
holdsAt(B, T1)	← initiates(A, B, T) ∧ happens(A, T) ∧ ¬ clipped(T, B, T1) ∧ T < T1.
¬ holdsAt(B, T1)	← initiallyFalse(B) ∧ ¬ declipped(0, B, T1).
¬ holdsAt(B, T1)	← terminates(A, B, T) ∧ happens(A, T) ∧ ¬ declipped(T, B, T1) ∧ T<T1.

Figure 2: Event Calculus predicates and axioms

The Event Calculus supports both deductive and abductive reasoning. *Deduction* uses the description of the system behavior together with the history of events occurring in the system and the domain independent axioms to derive the fluents that will hold at a particular point in time. Whilst deductive reasoning can be used to verify simple properties for a managed system, such as checking if a particular policy is applicable at a given point in time, or determine the set of access rights granted to a given user, the reasoning technique that is of particular interest to our work is *abduction*. Abduction can be used, given the descriptions of the behavior of the system, 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. Therefore, we define happens(A, T) as the abducible term in the language.

Because Event Calculus is expressed in First Order Logic (FOL), it supports all three modes of logical reasoning – deductive, inductive and abductive. This means that given an Event Calculus based representation, it is possible to automate the verification of a range of properties regarding the system. Deduction uses the description of the system behavior together with the history of events occurring in the system to derive the fluents[1] that will hold at a particular point in time. Induction aims to derive the descriptions of the system behavior 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. Abduction can be used, given the descriptions of the behavior of the system, 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. More formally this can be described as the derivation of a set of assertions Δ, given a system description D and a goal sentence G, such that $(D \cup \Delta) \vdash G$ and $(D \cup \Delta)$ is consistent. The set Δ should only contain sentences that have been defined as *abducible*, where the exact set of abducible terms will be specific to the application domain. Since an inherent feature of the abductive procedure is to derive information that is "missing" in the system description (i.e. the abducibles) in order to

[1] A *fluent* is a property that varies over the lifetime of a system

satisfy the goal, the technique is able to deal with situations where only a partial definition of the initial system state is provided.

From an algorithmic perspective, the process of deriving the set Δ (called the *abductive inference procedure*) is usually composed of two phases – an *abductive phase* followed by a *consistency phase* which interleave with each other. In the first phase a set of temporary abducibles is generated and must checked for consistency with respect to the system description in the second phase before they can be added to the final solution. The listing below presents a pseudo-code description of a simple abductive proof procedure.

Assumptions:

- The system description, D consists of rules of the form **A ← L1 . L2 . .. . Ln**, where head of the rule A is a positive atom and the clauses in the body of the rule L1 .. Ln, are atoms, Ai (or ¬Ai).

- An abductive solution is obtained by calling the function, **generateAbducibles(Goal, Delta)**, where the Goal is an atom and Delta will contain the set of abducibles that together with the system description satisfy the goal.

- Abducible atoms, Δi, are defined as such using the predicate **abducible(Δi)**.

- The consistency properties of the system are defined using rules of the form **ic ← P1 . P2 . .. . Pn**, were the conjunction of terms in the body of the rule represent the properties that hold when there is an inconsistency

Pseudo-code:

```
generateAbducibles(G, DELTA) {
        if (G is an atom that exists as a fact in the system description)
           Return the solution DELTA.
           END.
        else if (abducible(G) is a fact in the description)
           if not((D 4 DELTA 4 G) δ ic)        // CONSISTENCY CHECK
                  Add G to the solution DELTA.
                  Return the solution DELTA.
                  END.
           else
                  Return the solution DELTA
                  END.
        else if (G is an atom at the head of a rule, G ← L1 . .. . Ln)
           for each clause in the body of the rule, Li
                  Call generateAbducibles(Li, DELTA)
}
```

Over the years, many abductive inference procedures have been proposed. Work by Eshigi and Kowalski [7], Kakas and Mancarella [8], together with the techniques SLDNFA [9], IFF [10] and ACLP [11] are some of the most influential examples in this arena. However, none of these proposals resulted in an efficient implementation that could serve as a general solver for abductive logic programs. The ASystem, developed by Van Neuffelen [12], addresses the need for an abductive proof procedure for use in a framework for policy analysis and refinement because it has been shown to provide an efficient implementation of a general abductive inference procedure. Specifically, the ASystem reduces the goal formula that is presented to the abductive inference procedure into more basic formulas and then uses a constraint solver to evaluate these formulas and infer the set of abducibles that will satisfy the original goal. The idea of reducing a problem so that multiple external solvers can be applied to produce a solution is inspired by the Abductive Constraint Logic Programming (ACLP) technique described in [11].

The current implementation of the ASystem integrates a finite domain constraint solver and a (dis)equality constraint solver, and since this is done in a modular way it is possible to extend the system's capabilities or improve them as better solvers are developed. Also, the use of constraint solvers allows the ASystem to evaluate the integrity constraints that ensure the consistency of the overall solution more efficiently.

One significant limitation of the ASystem is that it does not deal with problem specifications that include cyclic dependencies and in these cases can get trapped in a loop. We are able to avoid this being a problem by constraining our formal specification to a subset of first order logic known as *stratified logic*. A stratified program is one where it is possible to order the clauses such that for any clause containing a negated literal in its body, there is a clause later in the program that defines the negated literal. Another way of describing stratified theories makes use of directed dependency graphs. These are graphs that comprise a node for each predicate symbol appearing in the program and a directed edge from the node representing any predicate that appears in the body of a clause to the node representing the predicate defined in the head of the clause. The edges are labelled positively or negatively, where a negative symbol indicates that the predicate at the tail end of the edge appear in negated form in a clause of the program. Using this technique, a program is stratified if the dependency graph contains no cycles having a negative edge. Informally, stratified logic programs are those that have a constrained use of recursion and negation. This is desirable because numerous studies have identified stratified logic as a class of first order logic that supports logic programs that are decidable in polynomial time [13].

As will be shown in our previous work, Event Calculus can be used to represent the dynamic behavior of a policy managed system and abductive reasoning provides a means of deriving the sequence of events required to satisfy particular properties of a managed system. In this context, abductive reasoning provides similar capabilities to model checking [], but has the advantage of being able to deal with incomplete specifications of the initial state. Additionally, since the Event Calculus representation of a policy managed system is in first order logic, deductive reasoning can be used to check static properties of the system. This makes the use of abductive reasoning in conjunction with the Event Calculus preferable to model checking for the types of analysis required in a policy analysis and refinement framework.

B.2.2. Goal-based Approach to Policy Refinement

The first part of our policy refinement process is a technique for refining high-level goals into concrete achievable goals, often referred to as System Requirements. The next part of the refinement process maps these system requirements to specific modules/operations that are available within the system. In this process, each high-level goal is refined into sub-goals, forming a refinement hierarchy where the dependencies between goals at different levels of refinement are based on the type of goal decomposition used (AND/OR). Additionally there can be dependencies between goals in different hierarchies. The refinement process involves following a particular path down the hierarchy, at each stage 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.

TABLE I
APPLICATION-INDEPENDENT GOAL ELABORATION PATTERNS

Ref	Goal	Subgoals
GP1	$P \rightarrow \lozenge Q$	$(P \rightarrow \lozenge R) \wedge (R \rightarrow \lozenge Q)$
GP2	$P \rightarrow \lozenge Q$	$(P1 \rightarrow \lozenge Q) \wedge (P2 \rightarrow \lozenge Q) \wedge (P \rightarrow P1 \vee P2)$
GP2'	$P \rightarrow \lozenge Q$	$(P \rightarrow P1 \wedge P1 \rightarrow \lozenge Q) \vee (P \rightarrow P2 \wedge P2 \rightarrow \not\subset Q)$

KAOS [3] 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 $P \rightarrow \lozenge Q$ (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.

Whilst the work presented in [3] only considers domain-independent elaboration patterns, it is also possible to use these domain-independent results to develop specialised patterns that apply to particular application domains. For example, Figure 3 shows the definition of a goal refinement pattern for a user login application.

This goal hierarchy states that the high-level goal of the application is that when a user attempts to login to an application (i.e. loginReq(user, application) occurs) then the loggedIn(user, application) goal is eventually achieved. The next level of the hierarchy shows that in order to satisfy the high-level goal it is first necessary to satisfy the sub-goals that the user has been authenticated and authorised to login to the application (authenticated(user) . allowed(application, loggedIn, user)). These two sub-goals can be further refined, resulting in two operational goals. The first states that if there is an AuthenticationService (as) which has a status indicating that the user is authorised then the goal authenticated(user) is satisfied. The other operational goal states that if there is an AccessController (ac) which is responsible for the application (ac.mgdObj = application) and the status of this access controller allows the user to login, then the goal allowed(application, loggedIn, user) is satisfied.

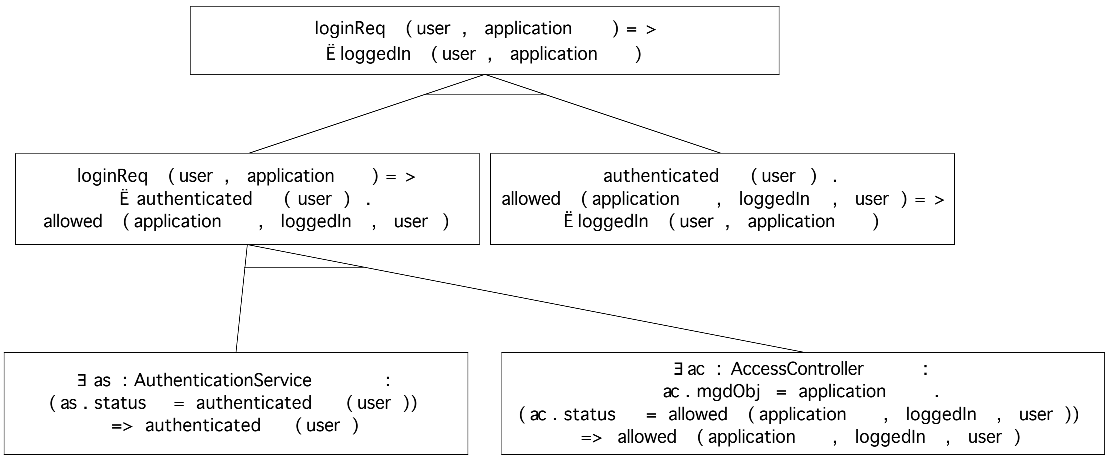

loginReq (user , application) = > Ë loggedIn (user , application)
loginReq (user , application) = > Ë authenticated (user) . allowed (application , loggedIn , user)
authenticated (user) . allowed (application , loggedIn , user) = > Ë loggedIn (user , application)
∃ as : AuthenticationService : (as . status = authenticated (user)) => authenticated (user)
∃ ac : AccessController : ac . mgdObj = application . (ac . status = allowed (application , loggedIn , user)) => allowed (application , loggedIn , user)

Figure 3: Application-specific goal elaboration pattern

Having defined this goal hierarchy, an administrator can reuse it to directly obtain the operational goals for a login procedure in any login application. This is possible because the refinement can be proven to be correct (using the proofs for the domain-independent patterns), and therefore it is no longer necessary to show the intermediate goals in the elaboration process when using it.

Being able to specify such application specific elaboration patterns allows users to translate high-level requirements into system requirements by simply instantiating the pattern with the information relevant to their particular application. A particular strength of KAOS is that, because the elaboration pattern includes a formal proof, the user can be confident that this transformation is correct. An additional advantage of the KAOS approach is that it is based on a formal notation, thus providing the possibility of using automated reasoning techniques for the goal elaboration process.

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 $\rightarrow$ 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 behavior, 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 behavioral specification notation and second, because it is possible to translate from the Statechart specification to the underlying formalism.

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:

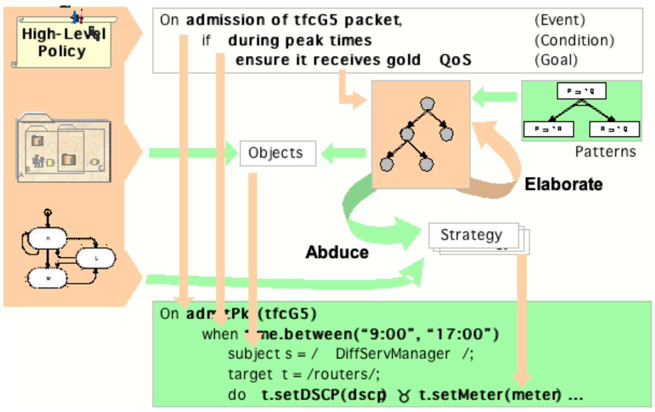
> **Strategy** AchievedGoal
>
> **OnEvent** Events derived from transitions with system events.
>
> **DerivedActions** Actions derived from transitions with operations.
>
> **Constraints** Constraints derived from guards.

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:

- 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.
- 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.
- 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 [14]. 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. Combining this approach to identifying the concrete objects with the goal elaboration and strategy derivation techniques, the overall policy refinement process can be summarized as follows.



```
On admission of tfcG5 packet,
    if  during peak times
        ensure it receives gold  QoS
```

```
On admitPkt (tfcG5)
    when time.between("9:00", "17:00")
        subject s = /  DiffServManager  /;
        target  t = /routers/;
        do  t.setDSCP(dscp) ⋎ t.setMeter(meter) ...
```

Figure 4: Overview of Policy Refinement Process

The user provides the high-level policy they are interested 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. 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. 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 4).

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.

B.3. Formal Representation of Policy Managed Systems

B.3.1 Example Application Scenario

In this report, we use an example application taken from the domain of DiffServ QoS management [15]. Specifically, we identify goals, strategies and policies in the context of the TEQUILA project [16]. TEQUILA uses the DiffServ mechanism, together with Multi-Path Labelled Switching (MPLS) [17] to provide a network that can dynamically adapt to meet the demands of varying network traffic. This adaptation is performed using a combination of online and offline techniques – from network dimensioning calculations that determine the upper/lower bounds of various network parameters based on the Service Level Agreements (SLAs) and traffic forecasts; to dynamic resource management and route management modules that make real-time changes to the router configuration to meet sudden variations in traffic.

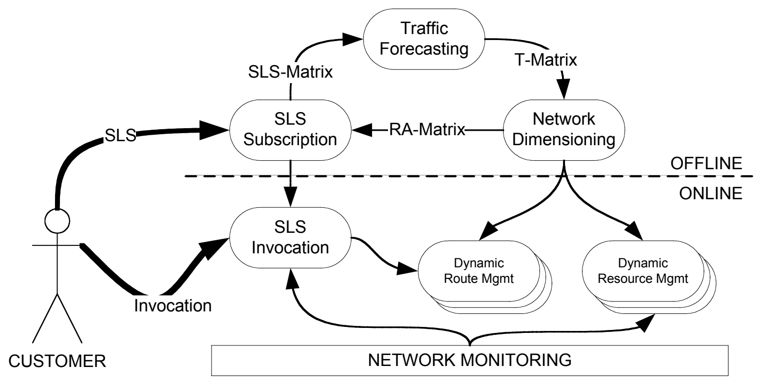


Figure 5: TEQUILA DiffServ QoS Management Framework

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 5 shows a logical representation of this architecture. The TEQUILA framework has been previously presented [18], 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.

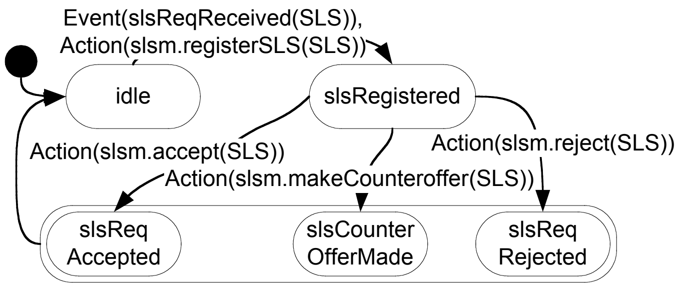


Figure 6: Service Level Specification Subscription (SLS-S) Module

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 6. A more detailed description of the subscription admission control algorithm can also be found in [19].

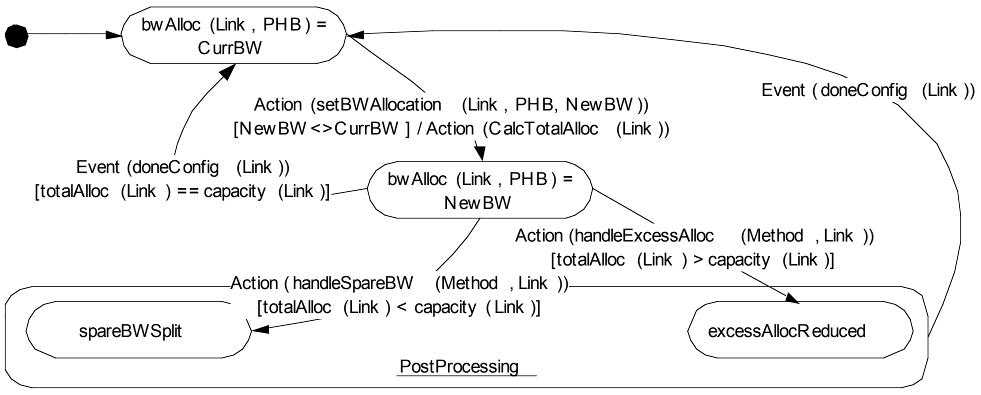

bwAlloc (Link , PHB) = CurrBW
Action (setBWAllocation (Link , PHB, NewBW))
[NewBW <> CurrBW] / Action (CalcTotalAlloc (Link))
Event (doneConfig (Link))
Event (doneConfig (Link))
[totalAlloc (Link) == capacity (Link)]
bwAlloc (Link , PHB) = NewBW
Action (handleExcessAlloc (Method , Link))
[totalAlloc (Link) > capacity (Link)]
Action (handleSpareBW (Method , Link))
[totalAlloc (Link) < capacity (Link)]
spareBWSplit
excessAllocReduced
PostProcessing

Figure 7: Dynamic Resource Management (DRsM) Module

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. The statechart representation of the behavior of the DRsM module is shown in Figure 7.

B.3.2 Managed Object Representation

In our view of a policy-based system, managed objects are organised into logical groups and are defined by an associated type. When adding a managed object to the system, the administrator must specify its type and define where the object is to be placed in the organisational model used in the system. In order to represent the organisational model of the managed objects, the administrator defines a domain hierarchy. As described previously, domains logically group objects together and can be used in policy specifications to denote that a policy applies to multiple objects. Additionally, domains allow the coupling between policies and managed objects to be loosely defined, effectively acting as a dynamic binding mechanism. Each type of managed object in a system can be described using a static model, consisting of the attributes and operations supported by the type; and a dynamic model that specifies the runtime behavior of objects of this type. We assume that the administrator specifies the type information using UML.

Given this description, a formal representation of managed objects must include the organisational model specified in terms of domain hierarchies; together with the static and dynamic models of the managed object. In this section we describe how this can be done by using the Event Calculus in conjunction with the predicates and functions shown in Table II.

B.3.2.1 Organisational Model

As mentioned, the organisational model of the system is defined in terms of a domain hierarchy. Domains are a means of grouping objects and membership of a domain is explicit and not defined in terms of a predicate on object attributes. Domains can be organised into a hierarchy by including one domain in another, but a sub-domain is not a subset of its parent domain, in that an object included in a sub-domain is not a direct member of the parent domain. Also, an object or sub-domain may be a member of multiple parent domains i.e. domains can overlap.

TABLE II
FUNCTIONS AND PREDICATES FOR REPRESENTING MANAGED OBJECTS

Function Symbol	Description
pot_state(Obj, Attr, Value)	Used to represent a potential state of a managed object when defining its behavioural model. The actual state of a managed object holds when a policy performs a management operation (see state fluent – Table 3.2).
op(Obj, OperationName, [Parameters])	Used to denote the management operations specified in a policy function or event (see below)
doAction(op(ObjTarg, OpName, [Parms]))	Represents the event of the action specified in the op term being performed on the target object, ObjTarg.

Predicate Symbol	Description
mgdObj(Obj, ClassName)	Used to specify that Obj is an object in the system, with the type denoted by ClassName.
attr(ClassName, AttrName, Type)	Defines that the class ClassName has an attribute called AttrName of type, Type.
domain(Obj)	Defines that Obj represents a domain. In order to indicate that a domain is a specialisation of an object, we also define the following rule: mgdObj(Obj, 'classDomain') $\leftarrow$ domain(Obj).
isMember(Obj, Dom)	Holds if the object, Obj, is a member of the Domain, Dom.
subDomain(Child, Parent) $\leftarrow$ domain(Child), domain(Parent),	Holds if the domain represented by Child is a sub-domain of Parent. The body of the rule is used to

isMember(Child, Parent), not(Child=Parent). subDomain(Child, Parent) ← domain(Parent), domain(Child), domain(IntParent), not(IntParent=Parent), not(IntParent=Child), isMember(IntParent, Parent), subDomain(Child, IntParent).	ensure that there are no cyclic relationships in the domain structure.
isDerivedMember(Domain, Object) ← mgdObj(Object, _), isMember(Object, Domain). isDerivedMember(Domain, Object) ← domain(Domain), mgdObj(Object, _), subDomain(SubDom, Domain), isDerivedMember(SubDom, Object).	Used to determine membership of a domain across the entire domain structure. This first rule identifies all those objects that are direct members of the domain, Dom. The second rule recursively identifies those objects that are members of sub-domains of the domain, Dom.

We define the predicate domain(DomainName) to represent a domain where the argument is a unique identifier. The absolute domain path expression for an object in the domain hierarchy (which take the form /dom1/…/obj) is assumed to be unique. Domain membership is represented using the isMember(ObjName, DomainName) predicate where the first argument is the name of the object to be added to the domain named by the second argument. Finally, managed objects are denoted by the mgdObj(ObjName, ClassName) predicate, where the first argument is a unique identifier for the managed object instance and the second argument is the name of the managed object class.

B.3.2.2 Static Model

The static model of a managed object is represented by its class definition, which specifies a class name, a set of attributes, and a set of methods. As shown in the example in the previous section, the mgdObj/2 predicate can be used to represent a managed object instance of a particular type. In order to represent the attributes of a class, we define a predicate attr(ClassName, AttrName, Type). The parameters of this predicate are used to specify that a given class (ClassName) has an attribute called AttrName of a particular type (Type).

For the purposes of policy-based management, we are only interested in those methods which are part of the management interface of the object. Since this information is also specified in the dynamic model of the object, in order to avoid duplication of information in the formal representation, we do not specify a stand-alone representation of the methods of a managed object. Instead, as will be described in the next section, we define predicates that allow a user to query the formal specification to determine the operations of a managed object class.

B.3.2.3 Dynamic Model

The dynamic model of a managed object describes it's the run-time behaviour in terms of the changes in state caused by performing the operations specified in the management interface of the object. For each type of managed object, the administrator can use an UML state chart to specify the state changes that occur when management operations are performed.

In order to model the behaviour of these operations, it is necessary to specify the pre- and post-conditions for each operation. 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(A, B, T) and terminates(A, B, T) predicates, according to the following schema:

```
initiates(doAction(ObjSubj, op(ObjTarg, Action, Parms)), PostTrue, Tm) ←
  PreCondition . mgdObj(ObjTarg, ClassName).
```

```
terminates(doAction(ObjSubj, 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(B, T) predicates. The mgdObj(ObjTarg, ClassName) predicate in the body of each rule indicates that this rule defines an operation for the type, ClassName.

The state of a managed object is represented by a conjunction of (Object, Attribute, Value) triplets. To reflect that the state chart is a representation of the potential states of the system, the PostTrue and PostFalse fluents are defined using pot_state(Object, Attr, Value) terms. In a policy managed system, a potential state translates into an actual state if there is an obligation to perform an action that defines the potential state as a post-condition. This is explained further in the description of the formal representation of the policy enforcement model (Section 3.4.6)

Building on the example shown previously, it is possible to illustrate the use of these rules for modelling system behaviour. Consider the SLS-S module that has the behaviour shown by the statechart in Figure 6. It is possible to transform this state chart into the Event Calculus notation presented previously where the action shown on each transition arrow is the operation being performed; for transitions between different states, the old values of any attributes that change become the PostFalse fluents; the new values of the attributes become the PostTrue fluents; and the current state values, together with any guards, 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, the state chart can be represented in the formal language as shown below.

```
1-      initiallyTrue(pot_state(Obj, status, 'init')) ←
                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).
```

B.3.3 Policy Representation

The next step in describing the formal language is to determine the features of policy rules that need to be expressed in the formalism. Since the scope of this thesis is limited to security authorisation and management/behavioural policies, we only consider positive authorisation, negative authorisation, obligation and refrain policies.

Each type of policy contains a number of elements – Subjects, Targets, Actions, Constraints and, in the case of obligation policies, Events. In order to provide a complete formal representation of policies, the information contained in each of these elements must be included in the formalism. The detailed formal representation of each of the elements is presented in [20]. In this section we provide an overview of the formal representation of the different types of policy rule and then proceed to model how enforcing a policy affects the state of the managed system. Therefore, we conclude this section with a description of the logic rules used to model the effect of the policies on the state of the managed system. Table III shows the predicates and functions used in our formalism.

TABLE III
FUNCTIONS AND PREDICATES FOR REPRESENTING POLICIES

Function Symbol	Description
allow(ObjSubj, op(ObjTarg, OpName, Parms))	Represents the permission granted to a subject, ObjSubj, to perform the action, OpName, on the target, ObjTarg. Parms are included in case OpName is overloaded
deny(ObjSubj, op(ObjTarg, OpName, Parms))	Used to denote that the subject, ObjSubj, is denied permission to perform the action OpName on the target, ObjTarg.
Oblig(ObjSubj, op(ObjTarg, OpName, Parms))	Denotes that the subject, ObjSubj, should perform the action specified in the operation term on the target, ObjTarg.
refrain(ObjSubj, op(ObjTarg, OpName, Parms))	Denotes that the subject, ObjSubj, should not perform the action specified in the operation term on the target, ObjTarg.
state(Obj, Attr, Value)	Represents the value of a managed object attribute. This fluent holds when an obligation to perform an action with the post-condition pot_state(Obj, Attr, Value) is enforced.
sysEvent(E)	Represents a system event. Used in the definition of obligation policies.
clocktick(H,M,S)	Event that represents the real-world time having the value H:M:S. E.g using this function, 2pm would be

	represented as clocktick(14,0,0)
done(Policy, Subj, Op)	Event that occurs when an obligation policy, Policy, has been enforced by the subject performing the specified operation.

Predicate Symbol	**Description**
requestAction(Policy, Subj, Op, T)	Denotes that the specified policy made a request for the subject to perform the operation at time, T.

TABLE IV
FUNCTIONS AND PREDICATES FOR REPRESENTING GOALS

Function Symbol	**Description**
achieve(P, Q)	Represents a goal expressed using the temporal logic formula, P υ ⊄Q, and means that if P holds Q will hold at some point in the future.
maintain(P, Q)	Represents a goal expressed using the temporal logic formula, P υ ≈Q, and means that if P holds that Q will hold at all points in the future.
cease(P, Q)	Represents a goal expressed using the temporal logic formula, P υ ⊄¬Q, and means that if P holds ¬Q will hold at some point in the future (i.e. Q will cease to hold).
avoid(P, Q)	Represents a goal expressed using the temporal logic formula, P υ ≈¬Q, and means that if P holds ¬Q will hold at all points in the future (i.e. Q will be avoided).
until(P, Q, R)	Represents a goal expressed using the temporal logic formula, P υ Q U R, and means that if P holds Q will hold for all points until R holds.
unless(P, Q, R)	Represents a goal expressed using the temporal logic formula, P υ Q W R, and means that if P holds Q will hold unless R holds.
and(P1, P2)	Conjunction of the two goal terms, P1 and P2. Indicates that they both hold at a given point in time.
goalProperty(P)	Used to indicate that P is a property that can be used as part of a goal definition.

Predicate Symbol	**Description**
holdsGoal(P, T)	Defines that goal P holds at time, T. The time argument is used to determine the relative order of a set of goals terms.
ptnGoal(G)	The high-level goal expression for a goal elaboration pattern. Typically, this would be instantiated as: goalPtn(achieve(P,Q)) ← ... subgoalPtn ...
ptnSubGoal(SubGOal)	Represents a sub-goal expression in a goal elaboration pattern. It is used in a body of a rule that defines the pattern.
missingGoalProperty(G)	Denotes a property G that must be defined by the user.

B.3.4 Goal Representation

The formal language presented thus far can be used to represent policy managed systems and allows such specifications to be analysed. However, in order to support policy refinement the formalism must also capture information about the high-level goals the administrator wishes to achieve and how these relate to lower-level goals that can actually be achieved by the systems.

The KAOS goal elaboration technique presented in the Chapter 2 provides a formal technique for deriving low-level goals from high-level ones. This technique uses goal elaboration patterns (proven decompositions of high-level goals) which facilitate the derivation of low-level goals that provably satisfy the original high-level goal. In the KAOS technique, goals and goal elaboration patterns are expressed using temporal logic formulae and therefore in order to include this information in our formalism we must provide support for expressing the temporal logic operators such as eventually, always, unless and until (see Section 2.2). In this section we describe how the predicates and functions defined in Table can be used to represent each of these types of information, together with the rules that allow automated reasoning to be used to derive the results required for policy refinement.

B.3.4.1 Basic Goals

In the KAOS technique, a goal is represented by its name, informal description and formal definition. In the broadest sense, there are four types of goal, referred to as *achieve*, *cease*, *maintain* and *avoid* goals, and each of these types can be characterised by a temporal logic formula as follows:

- ***Achieve goals*** specify the requirement that given some initial conditions, P, the system should eventually satisfy some property, Q. This is represented in temporal logic as: P υ ⊄ Q.
- ***Cease goals*** are used to state the requirement that given some initial conditions P, the system should eventually satisfy the negation of some property, Q. Formally this is stated as: P υ ⊄ (¬Q).
- ***Maintain goals*** are used to state the requirement that given some initial conditions, P, the system should always satisfy some property, Q, in the future. The temporal logic representation of this is: P υ ≈ Q.
- ***Avoid goals*** specify the requirement that given some initial conditions P, the system should always satisfy the negation of a property, Q. Formally this is stated as: P υ ≈ ¬Q, which is logically equivalent to P υ ¬ ⊄ Q.

```
|T1: holdsgoal(Q, T1)  ← holdsGoal(achieve(P, Q), T),
                         holdsGoal(P, T), T1 > T.

|T1: ¬ holdsGoal(Q, T1) ← holdsGoal(cease(P,Q), T),
                          holdsGoal(P, T), T1 > T.

...T1: holdsGoal(Q, T1)  ← holdsGoal(maintain(P, Q), T),
              holdsGoal(P, T), T1 > T.

...T1: ¬ holdsGoal(Q, T1)  ← holdsGoal(avoid(P, Q), T),
                             holdsGoal(P, T1), T1 > T.
```

In each of these formal expressions, P and Q represent some property (or conjunction of properties) that the system must satisfy initially (P) and in the future (Q). At the lowest level, these properties should correspond to potential states of the managed system (i.e. states defined in the behavioural model of the managed objects), however at high-levels of abstraction properties can be represented by names of other goals that must be satisfied.

Each type of goal can be represented in the formal language using the terms achieve(P,Q), cease(P,Q), maintain(P,Q) and avoid(P,Q) respectively. These terms can be used in conjunction with the holdsGoal(G, T) predicate to specify rules that determine the truth of a given goal (see below). The holdsGoal(G, T) predicate is similar to the holdsAt(B, T) predicate defined as part of the Event Calculus. It is necessary to define this new predicate, rather than simply use holdsAt(...), because the temporal logic definitions used in goal elaboration do not have a direct correspondence to the Event Calculus axioms.

```
holdsgoal(Q, T1)  ← holdsGoal(unless(P, Q, R), T),
                    holdsGoal(P, T),
    ¬ holdsGoal(R, T1), T1 > T.

holdsgoal(Q, T1)  ← holdsGoal(until(P, Q, R), T),
                    holdsGoal(P, T),
        holdsGoal(R, T2), T2 > T1 > T.

holdsGoal(Q, T)   ← holdsGoal(and(P,Q), T),
        holdsGoal(P, T).

holdsGoal(P, T)   ← holdsGoal(and(P,Q), T),
                    holdsGoal(Q, T).
```

Each of the rules above provides an implicit definition of the eventually holds and always holds temporal operators. However, goal descriptions can also involve the other temporal operators, unless (if P then Q unless R) and until (if P then Q until R). Additionally, it is necessary to be able to specify conjunctions of goals. The rules defined in Listing 3.12 extend the formal language to allow these additional operators to be included in a goal specification.

In the formal language, we define the function goalProperty(Name) to represent a named property that might be used in a goal definition. This function can be used in a holdsGoal(G, T) predicate in the head of a rule, where the body specifies the goal expression that satisfies the property. As mentioned previously, higher-level goal properties can be defined in terms of other goals. This type of goal property will be represented in the formal language by using holdsGoal(Goal, T) predicates in the body of the rule. In the TEQUILA framework's SLS subscription example, the administrator might specify a high-level goal stating that all incoming SLS requests must be processed by the SLS-S module. In this situation, the administrator might want to represent the domain knowledge that the high-level goal "Process SLS Request" can be represented by the two sub-goals "Register SLS" and "Accept SLS". The formal representation of this decomposition is shown below:

```
holdsGoal(goalProperty(process_sls), T) ←
        holdsGoal(achieve(receive_sls, register_sls), T),
        holdsGoal(achieve(register_sls, accept_sls), T).
```

In the case of an operational goal, the body of the rule defines the system state that must hold for the property to be satisfied, using holdsAt(pot_state(Obj, Attr, Value), T) predicates. For example, in the scenario above, the low-level goal for registering an SLS might be defined as the state where the SLS-S manager is in the 'registered' status. This would be represented in the formal notation as:

```
holdsGoal(goalProperty(register_sls), T) ←
        mgdObj(SLSMGR, classSLSMgr),
        holdsAt(pot_state(SLSMGR, status, 'slsRegistered'), T).
```

Alternatively, certain operational properties might be satisfied through the occurrence of an event. In this case, the body of the rule is defined using a happens(A, T) predicate. In the example presented above, if the receive_sls goal property is satisfied when a receiveSLS event occurs, this would be represented in the formal language as:

```
holdsGoal(goalProperty(receive_sls), T) ←
        happens(sysEvent(receiveSLS), T).
```

B.3.4.2 Goal Elaboration Patterns

In addition to the goals specific to a particular managed system, if we are to provide some automated reasoning support for decomposing goals into sub-goals, the formal language must also capture the goal elaboration patterns. A goal elaboration pattern is represented by a goal, together with a set of sub-goals that satisfy the goal. Formally, the relationship between the higher-level goal, G, and its sub-goals, G1 … Gn, can be expressed as: G1, …, Gn δ G.

We introduce the predicates ptnGoal(Goal) and ptnSubGoal(SubGoal) which respectively represent the high-level goal and the sub-goals of an elaboration. The Goal/Sub-goal definitions for these predicates would be expressed using the goal functions described in the previous section. For example, the domain independent goal elaboration pattern (P υ ⊄R), (R υ ⊄Q) δ (P υ ⊄Q), would be specified in this notation as:

```
ptnGoal(achieve(P, Q)) :- ptnSubGoal(achieve(P, R)),
                ptnSubGoal(achieve(R, Q)),
                missingGoalProperty(R).
```

The purpose of decomposing a goal is to identify some new, lower-level, properties in the sub-goals expressions such that the original high-level goal can be satisfied. At each stage of the elaboration process, the user will be required to define these new properties in terms of other goals or as a required system state. Therefore, when specifying a goal elaboration pattern, the predicate missingGoalProperty(Name) is used to identify the properties of the sub-goal definitions that need to be provided by the user. In order to provide a means of automatically identifying the sub-goals and missing goal properties for a given goal, we extend the set of abducible terms in the formal language to include ptnSubGoal(…) and missingGoalProperty(…). This means that given a goal that matches the head of an elaboration rule of the form defined above, the system can abduce the sub-goals and missing goal elaboration patterns required for a correct decomposition of the goal.

```
1   ptnGoal(achieve(loginReq(user, appl), logged_in(user, appl))) :-
      ptnSubGoal(achieve(loginReq(user,appl), authenticated(user))),
      ptnSubGoal(achieve(loginReq(user,appl), authorised(user, appl))),
      ptnSubGoal(achieve(and(authenticated(user), authorised(user, appl)),
              logged_in(user, appl))),
      missingGoalProperty(user), missingGoalProperty(appl).

2   holdsGoal(goalProperty(authenticate(user), T) :-
            mgdObj(AS, classAuthenticationServer),
            holdsAt(pot_state(AS, status, authenticated(user)), T).

3   holdsGoal(goalProperty(authorise(user, appl), T) :-
            mgdObj(AC, classAccessController),
            holdsAt(pot_state(AC, mgdObj, appl), T).
            holdsAt(pot_state(AC, status, allowed(appl, loggedIn, user)), T).
```

The KAOS technique also supports the use of application-specific goal elaboration patterns. For example, in section 2.2 we presented an elaboration pattern for a user login application which stated that for a user to be logged in, the goals for authentication and authorisation must be satisfied (see Chapter 2, Section 2.2). The formal representation of this pattern is shown above. Here, Line 1 defines the elaboration pattern which required that the administrator specify the user and application definitions for the particular situation in which the pattern is being applied. The remaining rules define the system state required to satisfy the authentication and authorisations sub-goals specified in the elaboration pattern.

In this section we extended the formal language to include representations of both goals and goal elaboration patterns. The representation of goal elaboration patterns allows abductive reasoning to be used to determine the possible decompositions of a given high-level goal. Having selected a suitable decomposition, the user must then provide a definition for any missing properties in the sub-goal definitions. This process will be repeated until the sub-goals are expressed in terms of desired states of managed objects. We refer to such sub-goals as operational goals.

Once the low-level operation goals have been defined, it is possible to use this representation in conjunction with the system behaviour model to abduce the set of operations required to satisfy a particular goal. We refer to this set of operations as the *strategy* for achieving the goal.

B.4. Automated Policy Refinement

Before detailing the approach to policy refinement that we have developed, it would be useful to consider an example that illustrates the procedure that would be followed if an administrator were to refine a high-level goal into a policy manually.

This scenario illustrates how the TEQUILA framework responds to short-term changes in traffic from a particular customer. Taking the same example network as before, consider the situation where the network experiences a short-term increase in the traffic pertaining to the SLA described in the previous scenario. The network administrator wants to ensure that when this increase occurs during between 11am and 1pm and causes a network utilisation greater than 85% of the maximum allocation calculated by the ND module, the bandwidth allocation should be increased only by 10% and any spare capacity should be equally split amongst the PHBs. In this situation the Dynamic Resource Management (DRsM) module at each link along the route followed by the traffic would respond as follows:

- Detect the increase in traffic, and decide to raise an alarm if necessary
- On receiving an alarm relating to an increase in traffic, the DRsM must decide on the appropriate action to adapt to the increase. To help with this, the DRsM has the guideline values for maximum, minimum and congestion bandwidth allocations provided by the network dimensioning process.
- Configure the link/PHB with this new value and decide on how to allocate any spare link capacity amongst all the link/PHBs

In this process it can be seen that policies are involved at all three stages, since there are decisions to be taken regarding when to raise an alarm, how to determine the new bandwidth allocation, and how to distribute the spare link capacity. As before, in each of these cases the exact policy to be used depends on the goal that the administrator is interested in. In this scenario, the specific goal is that an alarm should be raised if the bandwidth utilisation for the link/PHB is above 85% of the maximum allocation specified by the ND module.

```
G5: Goal BWUtilIncreaseAlarmRaised
  FormalDef  linkBWUtilIncrease(utilValue, PHB)
        → ◊ alarmRaised(bwUtilIncr, [utilValue, PHB]).


S4: Strategy G5: BWUtilIncreaseAlarmRaised
  DerivedActions  monitor.raiseAlarm(alarmType, [alarmParms]).
```

As shown above, the strategy for achieving the basic goal of "alarm is raised" (G5) can be automatically abduced without the need for elaboration. Combining this strategy (S4) with the constraints, the following policy, that satisfies the overall goal, can be written:

```
P2: inst oblig /policies/networkMonitoring/increaseBWUtilAlarm {
     on   linkBWUtilIncrease(utilValue, PHB);
     subj  s = /routers/FromR1/ToR6/drsmPMAs/;
     targ  t = s.monitor;
     do    t.raiseAlarm(bwUtilIncr, [utilValue, PHB]);
     when  utilValue > 0.85 * s.ndMaxBWAlloc(PHB); }
```

For the remaining policy decisions of determining the new bandwidth allocation value and then allocating any spare capacity, the strategy derivation is not as straight forward. Here the high-level goal is to achieve the state "adapted configuration" when an alarm is raised. This can be stated as follows:

G6: **Goal** ConfigAdaptedForBWUtilIncrease
FormalDef alarmRaised(bwUtilIncr, [utilValue, PHB]) → ◊ configAdapted.

In this case the abductive analysis of G6 yields no strategy, so it is necessary to elaborate the goal further. As a first step, applying GP2' yields the sub-goals NewRPCRequested (G7) or CalculatedConfiguredNewBWAllocation (G8) and as shown in their formal definitions below, each leads to the high-level goal of ConfigAdaptedForBWUtilIncrease being satisfied.

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 under consideration, the high-level policy involves calculating and configuring a new bandwidth allocation, a goal represented by 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 8 shows the goal elaboration hierarchy for this elaboration process, indicating the applicable patterns at each stage. The details of each of the goals in this diagram are as follows:

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 $\rightarrow$ spareCapEquallySplit .

spareCapEquallySplit $\rightarrow \Diamond$ configAdapted.

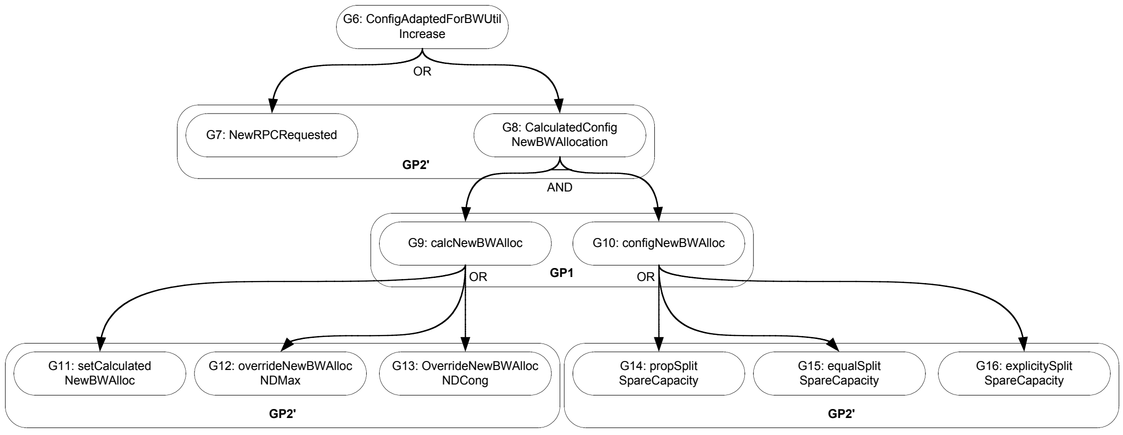
G16: **Goal** explicitySplitSpareCapacity

FormalDef configNewBWAlloc $\rightarrow$ spareCapExplicitlySplit([splitValues]) .

spareCapExplicitlySplit([splitValues]) $\rightarrow \Diamond$ configAdapted.

In this scenario we assume, the goals of the administrator are namely G11: setCalculatedNewBWAlloc and G15: equalSplitSpareCapacity. So, we are interested in the strategies for setting the new bandwidth allocation to the newly calculated value and splitting any 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 as shown in P3 below.

Notice that in this example, the abductive analysis results in a strategy that includes constraints which are derived from the guards defined in the state chart of the system behaviour. In order to ensure that the policy is valid with respect to the system behaviour, the administrator must include these constraints, together with any others that are manually mapped from the high-level policy, whenever the strategy is used. In the example, this is illustrated in the final policy, P3, which combines the constraint from the strategy with the time constraint from the high-level policy.



G6: ConfigAdaptedForBWUtil Increase
OR
G7: NewRPCRequested
G8: CalculatedConfig NewBWAllocation
GP2'
AND
G9: calcNewBWAlloc
G10: configNewBWAlloc
GP1
OR
OR
G11: setCalculated NewBWAlloc
G12: overrideNewBWAlloc NDMax
G13: OverrideNewBWAlloc NDCong
GP2'
G14: propSplit SpareCapacity
G15: equalSplit SpareCapacity
G16: explicitySplit SpareCapacity
GP2'

Figure 8: Traffic increase scenario goal elaboration hierarchy

```
S5: Strategy G11: setCalculatedNewBWAlloc && G15: equalSplitSpareCapacity
    OnEvent        alarmRaised(bwUtilIncr, [utilValue, Link, PHB])
    DerivedActions drsm.setBWAllocation(Link, PHB, CalcValue) ->
               drsm.handleSpareBW(equally, Link)
    Constraints    drsm.incrAllocBW(PHB, pct) < drsm.ndMaxBWAlloc(PHB).

P3: inst oblig /policies/adaptTrafficIncreaseAOLSLA_P1 {
      on   alarmRaised(bwUtilIncr, [utilValue, link, ef]);
      subj  s = /drsmPMA;
      targ  t = /routers/FromR1/ToR6/drsm/;
      do    t.setBWAllocation(link, ef, NewBWValue) ->
          t. handleSpareBW(equally, link);
      when  NewBWValue < t.ndMaxBWAlloc(ef) &&
          time.between('11:00', '13:00');
  }
```

Integrity constraints provide an elaboration tolerant way of specifying the constraints to be satisfied by different entities in a system. Additionally, they allow rules that define the consistency requirements of the system to be defined in a modular manner. This is advantageous since it allows the user to maintain the independence between rules that pertain to different aspects of a system. In the remainder of this section we show how integrity constraints can be used to enhance the policy refinement procedure in a number of different ways:

B.4.1. Avoiding Inconsistencies

It is important to ensure that policies derived by the refinement process are consistent with respect to other policies and the system itself. By defining inconsistency conditions as integrity constraints it is possible to ensure that the management operations derived as part of the policy refinement procedure do not cause any inconsistencies.

We have also made use of integrity constraints to guarantee that the objects in the derived strategies are of correct type. For example, to ensure that the setBWAllocation(Link, PHB, BW) operation is only performed by a DiffServRouter object on a DRsM object, we can specify the following constraints:

```
1     ic :- happens(doAction(R, op(_, setBWAllocation(_,_,_))), _),
          not(obj(R, cDiffServRouter)).
2     ic :- happens(doAction(_, op(D, setBWAllocation(_,_,_))), _), not(obj(D, cDRSM)).
```

Other types of consistency checks involve ensuring that policies do not conflict, and that appropriate constraints are derived so that the derived policy is only applicable if the actions specified are valid at the point of enforcement. In order to avoid simple modality conflicts, we can specify integrity constraints of the form:

```
1     ic :- happens(doAction(Subj, Op, _), T), not(holdsAt(pos(allow(Subj, Op)), T)).
2     ic :- happens(doAction(Subj, Op, _), T), holdsAt(pos(refrain(Subj, Op)), T).
```

Here the first constraint specifies that it is inconsistent for the derived policy to contain any actions that are not authorised. Similarly, the second avoids those actions that would be prevented by a refrain policy in the system. It is also possible to specify integrity constraints that avoid semantic conflicts, such as conflicts of duty. For example, in our example scenario, it would be incorrect to attempt to allocate two different bandwidth values for the same Link/PHB at the same time. This is captured by the following rule:

```
1    ic :- happens(doAction(_, op(_, setBWAllocation(Link,PHB,BW1))), T),
          happens(doAction(_, op(_, setBWAllocation(Link,PHB,BW2))), T),
          clp(BW1 <> BW2).
```

4.2. Deriving Parameter Values for Management Operations

The implementation of the policy refinement process is only able to derive the set of management operations (i.e. the strategy) for achieving a given goal. This process does not provide any suggestions for parameter values that can be used when performing these management operations. By defining integrity constraints that specify the valid ranges for particular managed operation parameter values, it is possible to include the derivation of these parameter values in the policy refinement process.

In most cases, these parameters are provided by the events that trigger the policy, or they calculated within the context of the subject enforcing the policy. However, there are cases where the parameter values can be determined by other factors, such as values computed using attributes of objects and resources. For example, in the case of the above scenario new bandwidth values are required for the setBWAllocation(Link, PHB, NewBW) operation. The following constraints specify limits on this parameter:

```
1    ic :- happens(doAction(_, op(DRSM, setBWAllocation(Link, PHB, NewBW)), _), T),
          happens(alarm(incrUtil, Link, PHB), T0), clp(T0<T),
          holdsAt(pos(state(DRSM, bwAlloc(Link, PHB), CurrBW)), T)),
          not(clp(NewBW > CurrBW)).
2    ic :- happens(doAction(_, op(DRSM, setBWAllocation(Link, PHB, NewBW)), _), T),
          happens(alarm(incrUtil, Link, PHB), T0), clp(T0<T),
          holdsAt(pos(state(DRSM, ndMAxBW(Link, PHB), NDMaxBW)), T)),
          not(clp(NewBW =< NDMaxBW)).
```

Constraint (1) above specifies that in the situation of an increased utilization alarm the new bandwidth value should be greater than the current one; and constraint (2) specifies that it should not exceed the max value defined by the network dimensioning module (ndMaxBW).

B.4.3. Utility Functions

It is possible that even after all the integrity constraints relating to consistency, subject/target selection and parameter value derivation are applied, there are still multiple derived policies that satisfy the high-level goal. In this situation it is necessary to compute some measure of the utility for each derived policy and have the system enforce the policy with the highest utility. We can define integrity constraints to determine a utility value associated with each management operation and then compute an average of these to determine the utility of a given policy.

Since the policy refinement procedure can generate multiple policies that satisfy a given goal, it is useful to have some mechanism for deciding on the most appropriate policy for a given situation. One approach for doing this would be to compute a utility value for each policy solution and then choose the policy that has the highest utility. Computation of these utility values can be achieved as part of the abductive inference procedure through the use of integrity constraints. For example, to specify that the utility of splitting any spare bandwidth proportionally is greater than using equal splitting if >50% of traffic is marked for expedited forwarding (EF), we have:

```
1    ic :- happens(doAction(_, op(DRSM, handleSpareBW(prop, Link)), Utility), T),
          holdsAt(pos(state(DRSM, bwUtil(Link, ef), gt(50%))), T), time(T),
          not(clp(Utility#=<75)).
```

```
2    ic :- happens(doAction(_, op(DRSM, handleSpareBW(equal, Link)), Utility), T),
        holdsAt(pos(state(DRSM, bwUtil(Link, ef), gt(50%))), T), time(T),
        not(clp(Utility#=<55)).
```

The above integrity constraints specify that splitting spare BW equally has utility up to 55%, whereas under the same conditions, splitting spare BW proportionally has utility up to 75%. This gives preference to policies where spare bandwidth is split proportionally.

To summarise, by means of advanced abductive methods and their integration with CLP [ACLP] we have been able to enhance considerably our policy refinement methods This enhanced refinement process not only ensures that derived policies are consistent, but also provides parameter values for management operations and utility values that can be used to rank the results. Numerical constraints are formalized in a simple way within the declarative representation of the policy allowing us to address optimization requirements with the integrated use of CLP constraint solving.

B.5. Related Work

There are few practical studies on policy refinement. Power [21] 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 [22]. 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 [23] 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.

B.6. 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 [24]. Stratified logic programs are decidable in polynomial time [13]. Additionally, it is possible to show that the formal representation satisfies the requirements of a unit-refutable theory [20], a class for which the abductive solution derivation has been shown to be computable in polynomial time [7].

B.7. Conclusions and Future Work

The work presented in this report has shown how to automate the refinement of policies through the use of abductive constraint logic programming whilst hiding the details of the underlying formal techniques from the user. The technique is not only able to generate the management operations that must be performed for achieving a given goal, but can also derive the parameter values for the operations and ensure that the derived operations are consistent with respect to the overall system. Additionally, the technique supports the calculation of utility values that can be used to determine a preference ranking between multiple refined policies.

Achieving this functionality 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 [25]) are adopted.

A key limitation of our approach is the dependence on the correctness of the integrity constraints in order to abduce refined policies. This means that the lack of a solution may simply be due to an error in the integrity constraint specification rather than because of a genuine limitation in the capabilities of the modelled system. In order to address this problem it will be necessary to investigate techniques for analysing the overall specification to ensure the consistency of integrity constraints. Together with enhancing the tool support for the policy refinement procedure, addressing this limitation will be the focus of our future efforts.

Acknowledgements

We acknowledge financial support for this work from the EPSRC (Grant Nos: GR/R31409/01 and GR/S79985/01), CISCO Systems Inc. and IBM Research. This deliverable reports on the work carried out under the visiting fellowship of Professor Kakas at Imperial College, London. This fellowship was supported by EPSRC (grant GR/T29246/01) under the project "Reasoning Techniques for Analysis and Refinement of Policies for Quality of Service Management.

References

1. Bandara, A.K., E.C. Lupu, and A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. in 4th IEEE International Workshop on Policies for Networks and Distributed Systems (Policy 2003). 2003. Lake Como, Italy: IEEE.

2. Moffett, J. and M.S. Sloman, Policy Hierarchies for Distributed Systems Management. IEEE Journal on Selected Areas in Communications, 1993. **11**(9 - Special Issue on Network Management): p. 1404-14.

3. Darimont, R. and A. van Lamsweerde, Formal Refinement Patterns for Goal-Driven Requirements Elaboration. 4th ACM Symposium on the Foundations of Software Engineering (FSE4), 1996: p. 179-190.

4. Bandara, A.K., E.C. Lupu, and A. Russo. A Goal-based Approach to Policy Refinement. in 5th IEEE International Workshop on Policies for Distributed Systems and Networks. 2004. IBM TJ Watson Research Centre, New York, USA: IEEE Press.

5. Kowalski, R.A. and M.J. Sergot, A logic-based calculus of events. New Generation Computing, 1986. **4**: p. 67-95.

6. Miller, R. 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., A. Kakas and F. Sadri, Editors. 1999, Springer. p. 452-490.

7. Eshghi, K. A Tractable Class of Abduction Problems. in International Joint Conference on Artificial Intelligence. 1993. Chambery, France: Morgan-Kaufmann.

8. Kakas, A. and P. Mancerella. Abductive Logic Programming. in International Workshop on Non-monotonic Reasoning and Logic Programming. 1990. Austin, Texas.

9. Denecker, M. and D. De Schreye, SLDNFA: an abductive procedure for normal abductive prorgamming. Journal of Logic Programming, 1998. **34**(2): p. 111-167.

10. Fung, T.H. and R.A. Kowalski, The IFF Proof Procedure for Abductive Logic Programming. Journal of Logic Programming, 1997. **33**(2): p. 151-165.

11. Kakas, A., A. Michael, and C. Mourlas, ACLP: Abductive constraint logic programming. Journal of Logic Programming, 2000. **44**(1-3): p. 129-177.

12. van Nuffelen, B. and A. Kakas. A-System : Programming with abduction. in Logic Programming and Nonmonotonic Reasoning (LPNMR 2001). 2001: Springer-Verlag.

13. Jager, G. and R.F. Stark, The Defining Power of Stratified and Hierarchical Logic Programs. Journal of Logic Programming, 1993. **15**(1 & 2): p. 55-77.

14. Damianou, N., et al. Tools for Domain-based Policy Management of Distributed Systems. in 8th IEEE/IFIP Network Operations and Management Symposium. 2002a. Florence, Italy.

15. Carlson, M., et al., An Architecture for Differentiated Services, in Network Working Group - RFC2475, *http://www.ietf.org/rfc/rfc2475.txt*. 1998.

16. Flegkas, P., P. Trimintzios, and G. Pavlou, A Policy-based Quality of Service Management Architecture for IP DiffServ Networks. IEEE Network Magazine Special Issue on Policy Based Networking, 2002. **16**(2): p. 50-56.

17. Rosen, E., A. Viswanathan, and R. Callon, Multiprotocol Label Switching Architecture, in Network Working Group - RFC3031, *http://www.ietf.org/rfc/rfc3031.txt*. 2001.

18. Trimintzios, P., et al., Service-driven Traffic Engineering for Intra-domain Quality of Service Management. IEEE Network Magazine Special Issue on Network Management of Multiservice, Multimedia, IP-based Networks, 2003. **17**(3): p. 29-36.

19. Mykoniati, E., et al., Admission Control for Providing QoS in IP DiffServ Networks: the TEQUILA Approach. IEEE Communications Magazine, 2003. **41**(1).

20. Bandara, A.K., A Formal Approach to Analysis and Refinement of Policies, in Computing. 2005, Imperial College London: London. p. 184.

21. Casassa Mont, M., A. Baldwin, and C. Goh, POWER Prototype: Towards Integrated Policy-Based Management. 1999, HP Laboratories Bristol: Bristol, UK.

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

23. Beigi, M.S., S. Calo, and D. Verma. Policy Transformation Technqiues in Policy-based Systems Management. in International Workshop on Policies for Distributed Systems and Networks. 2004. Yorktown Heights, New York: IEEE.

24. Apt, K.R., H.A. Blair, and A. Walker, Towards a Theory of Declarative Knowledge, in Foundations of Deductive Databases, J. Minker, Editor. 1988, Morgan Kaufmann: San Mateo, CA. p. 89-148.

25. DMTF, Common Information Model (CIM) Specification, version 2.2. 1999a.

26. M. Charalambides, P. Flegkas, G. Pavlou, A. Bandara, E. Lupu, A. Russo, N. Dulay, M. Sloman, J. Rubio-Loyola, "Policy Conflict Analysis for Quality of Service Management", in Proc. of 6th IEEE Int. Workshop on Policies for Distributed Systems and Networks, Sweden, June 2005.

27. G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. Artificial Intelligence Research, 4:19-36, 1996.

28. Y. Dimopoulos and A. C. Kakas. Logic programming without negation as failure. In Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, pages 369-384, 1995.

29. A.C. Kakas, R.A. Kowalski and F. Toni, "The Role of Abduction in Logic Programming", *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5:235-324, D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press, 1998.

30. M. Denecker and A.C. Kakas, Abduction in Logic Programming, in Computational Logic: Logic Programming and Beyond, LNAI Vol, 2407, pp. 402-437, Springer Verlag, 2002.

31. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, KR'98: Principles of Knowledge Representation and Reasoning, pages 86{97. Morgan Kaufmann, San Francisco, California, 1998.

32. Y. Dimopoulos and A. C. Kakas. Logic programming without negation as failure. In Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, pages 369-384, 1995.

33. A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentationtheoretic approach to default reasoning. Arti_cial Intelligence, 93:63-101, 1997.

34. P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Arti_cial Intelligence, 77:321-357, 1995.

35. A. C. Kakas, P. Mancarella, and P. M. Dung. The acceptability semantics for logic programs. In Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, pages 504-519, 1994.

36. H. Prakken and G. Sartor. A system for defeasible argumentation, with defeasible priorities. In International Conference on Formal and Applied Practical Reasoning, LNAI 1085, pages 510-524. Springer-Verlag, 1996.

37. Ehab Al-Shaer and Hazem Hamed, "Firewall Policy Advisor for Anomaly Detection and Rule Editing", IEEE/IFIP Integrated Management (IM'2003), 2003.

38. D. Chapman and E. Zwicky. *Building Internet Firewalls, Second Edition*, Orielly & Associates Inc.,2000.

39. W. Cheswick and S. Belovin. *Firewalls and Internet Security*, Addison-Wesley, 1995.

40. L.Qiu, G. Varghese and S. Suri, Fast Firewall Implementations for Soiftware and Hardware Routers, Proceedings of the 9th International Conference on Network Protocols (ICNP' 2001), 2001.

41. A. Mayer, A. Wool and E. Ziskind, Fang: A Firewall Analysis Engine, Proceedings of 2000 IEEE Symposium on Security and Privacy, 2000.

42. C.Basile, A.Lioy, Towards an algebraic approach to solve policy conflicts, *FCS'04: Foundations of Computer Security*, WOLFASI sub-workshop (Workshop On Logical Foundations of an Adaptive Security Infrastructure), Turku (Finland), July 12-13, 2004, pp. 319-338.

43. Ehab Al-Shaer and Hazem Hamed, Discovery of Policy Anomalies in Distributed Firewalls, In Proceedings of IEEE INFOCOM'04, March 2004.

44. Russo, A. *et al.* "An Abductive Approach for Analysing Event-Driven Requirements Specifications", in Proceedings of International Conference on Logic Programming, LNCS 2401, 22-37, 2002.

45. Gorgias: Argumentation and Abduction, http://www2.cs.ucy.ac.cy/~nkd/gorgias/.

46. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, F.Toni. *The KGP model of agency*. Proceedings of the 16th European Conference of Artificial Intelligence (ECAI'04),Valencia, Spain, August 24-27, 2004.

47. I. A. Letia, M. Acalovschi: Achieving Competence by Argumentation on Rules for Roles. ESAW 2004: 45-59.

APPENDIX A: Elevator Policy in GORGIAS

```
%==================================================================
%
%                ELEVATOR EXAMPLE POLICY
%=================================================================



% ==================================================================
% The program will return the moves of the elevator at all time
% points (0-9) under the policy by using the query:
%
%     elevator(example_name, Moves).
%
% where example_name is the name of the example.
% Currently two examples exist, example1 and example2.
%
% A new example (example3) can be added by using
% example(example3) :-  assert(rule(example3, happens(....), []),
%                       .....,
%                       assert(rule(example3, happens(....), []).
%
% ==================================================================

:- dynamic holds/2.
:- compile('gorgias').
:- compile('lpwnf').
:- compile('eres').


% ==================================================================
%           Temporal Reasoning (Auxiliary background Theory)
% ==================================================================

% Pressing a button inside the elevator (action press_button_in)
% initiates activated_button_in

rule(d1(T), initiation(activated_button_in(Floor), T),
[happens(press_button_in(Floor),T)]).

% Pressing a button at a floor (action press_button_floor) initiates
% activated_button_floor

rule(d2(T), initiation(activated_button_floor(Floor, Direction), T),
[happens(press_button_floor(Floor, Direction),T)]).

% When the elevator doors open (action open_door) at a floor,
% activated_button_in is terminated

rule(d3(T), termination(activated_button_in(Floor), T),
[happens(action(open_door, T), T)]) :-
       holds(elevator_position(Floor), T),
       holds(activated_button_in(Floor), T).

% When the elevator doors open (action open_door) at a floor,
% activated_button_floor is terminated
```

```
rule(d4(T), termination(activated_button_floor(Floor, Direction), T),
[happens(action(open_door, T), T)]) :-
      holds(elevator_position(Floor), T),
      holds(activated_button_floor(Floor, Direction), T).

% ACtion go_up initiates elevator_position(Floor) where Floor is one
% floor above the current position of the elevator

rule(d5(T), initiation(elevator_position(Floor1), T),
[happens(action(go_up, T),T)]) :-
      holds(elevator_position(Floor), T),
      directly_above(Floor, Floor1).

% Action go_up terminates elevator_position(Floor) where Floor is the
% current position of the elevator

rule(d6(T), termination(elevator_position(Floor), T),
[happens(action(go_up, T),T)]) :-
      holds(elevator_position(Floor), T).

% ACtion go_down initiates elevator_position(Floor) where Floor is
% one floor below the current position of the elevator

rule(d7(T), initiation(elevator_position(Floor1), T),
[happens(action(go_down, T),T)]) :-
      holds(elevator_position(Floor), T),
      directly_below(Floor, Floor1).

% Action go_down terminates elevator_position(Floor) where Floor is %
% the current position of the elevator

rule(d8(T), termination(elevator_position(Floor), T),
[happens(action(go_down, T),T)]) :-
      holds(elevator_position(Floor), T).

% ==================================================================
%          Basic Policy
% ==================================================================

% Elevator goes up when a button is pressed inside the elevator for a
% floor which is higher than the elevator's current position:

rule(r_up_in(Floor1, Floor2, T), action(go_up, T), []) :-
      holds(elevator_position(Floor1), T),
      holds(activated_button_in(Floor2), T), above(Floor2,Floor1).

% Elevator goes up when a button is pressed at a floor which is
% higher than the elevator's current position:

rule(r_up_floor(Floor1, Floor2, T), action(go_up, T), []) :-
      holds(elevator_position(Floor1), T),
      holds(activated_button_floor(Floor2, _), T),
      above(Floor2,Floor1).

% Elevator goes down when a button is pressed inside the elevator for
% a floor which is lower than the elevator's current position:

rule(r_down_in(Floor1, Floor2, T), action(go_down, T), []) :-
      holds(elevator_position(Floor1), T),
      holds(activated_button_in(Floor2), T), below(Floor2, Floor1).
```

```
% Elevator goes down when a button is pressed at a floor which is
lower than the elevator's current position:
rule(r_down_floor(Floor1, Floor2, T), action(go_down, T), []) :-
holds(elevator_position(Floor1), T),
holds(activated_button_floor(Floor2, _), T), below(Floor2, Floor1).

% Elevator' doors open when the elevator is at a floor and the button
for that floor is pressed from inside the elevator
rule(r_open_in(Floor, T), action(open_door, T), []) :-
holds(elevator_position(Floor), T), holds(activated_button_in(Floor),
T).

% Elevator' doors open when the elvator is at a floor and a button at
that floor is pressed:
rule(r_open_floor(Floor, Direction, T), action(open_door, T), []) :-
holds(elevator_position(Floor), T),
holds(activated_button_floor(Floor, Direction), T).

% Elevator may remain idle:
rule(r_idle(T), action(idle, T), []).

% Different actions are incompatible with each other:
complement(action(go_up, T), action(go_down, T)).
complement(action(go_up, T), action(open_door, T)).
complement(action(go_down, T), action(go_up, T)).
complement(action(go_down, T), action(open_door, T)).
complement(action(open_door, T), action(go_up, T)).
complement(action(open_door, T), action(go_down, T)).
complement(action(go_up, T), action(idle, T)).
complement(action(go_down, T), action(idle, T)).
complement(action(open_door, T), action(idle, T)).
complement(action(idle, T), action(go_down, T)).
complement(action(idle, T), action(open_door, T)).
complement(action(idle, T), action(go_up, T)).


% ==================================================================
%          Strategic Part
% ==================================================================

% The elevator' doors should not open when a button is pressed at a
floor for a certain direction,
% if the elevator is moving in the opposite direction:

rule(p_down_in_open_floor, prefer(r_down_in(Floor1, _, T),
r_open_floor(Floor1, up, T)), []).

rule(p_down_floor_open_floor, prefer(r_down_floor(Floor1, _, T),
r_open_floor(Floor1, up, T)), []).

rule(p_up_in_open_floor, prefer(r_up_in(Floor1, _, T),
r_open_floor(Floor1, down, T)), []).

rule(p_up_floor_open_floor, prefer(r_up_floor(Floor1, _, T),
r_open_floor(Floor1, down, T)), []).


% Elevator' doors should open when the elevator is at a floor and the
button of that floor is pressed
% from inside the elevator:
```

```
rule(p_open_in_up_in, prefer(r_open_in(Floor1, T), r_up_in(Floor1, _,
T)), []).

rule(p_open_in_up_floor, prefer(r_open_in(Floor1, T),
r_up_floor(Floor1, _, T)), []).

rule(p_open_in_down_in, prefer(r_open_in(Floor1, T),
r_down_in(Floor1, _, T)), []).

rule(p_open_in_down_floor, prefer(r_open_in(Floor1, T),
r_down_floor(Floor1, _, T)), []).


% Elevator doors should open when the elevator is at a floor and the
button for that floor is pressed
% from inside the elevator:

rule(p_open_floor_up_in, prefer(r_open_floor(Floor1, up, T),
r_up_in(Floor1, _, T)), []).

rule(p_open_floor_down_in, prefer(r_open_floor(Floor1, down, T),
r_down_in(Floor1, _, T)), []).


% Elevator doors should open at a floor if the button at that floor
is pressed for the direction
% in which the elevator is moving :

rule(p_open_floor_up_floor, prefer(r_open_floor(Floor1, up, T),
r_up_floor(Floor1, _, T)), []).

rule(p_open_floor_down_floor, prefer(r_open_floor(Floor1, down, T),
r_down_floor(Floor1, _, T)), []).


% Buttons pressed inside the elevator have higher priority than
buttons pressed at a floor:

rule(p_up_in_down_floor, prefer(r_up_in(Floor1, _, T),
r_down_floor(Floor1, _, T)), []).

rule(p_down_in_up_floor, prefer(r_down_in(Floor1, _, T),
r_up_floor(Floor1, _, T)), []).


% Buttons pressed at a floor above the elevator have higher priority
than buttons pressed at a floor
% below the elevator:

rule(p_up_floor_down_floor, prefer(r_up_floor(Floor1, _, T),
r_down_floor(Floor1, _, T)), []).


% Buttons pressed inside the elevator for a floor above the elevator
have higher priority than buttons
% pressed inside the elevator for a floor below the elevator

rule(p_up_in_down_in, prefer(r_up_in(Floor1, _, T), r_down_in(Floor1,
_, T)), []).
```

```
% Any action has higher priority than idle

rule(p_up_in_idle, prefer(r_up_in(_, _, T), r_idle(T)), []).
rule(p_up_floor_idle, prefer(r_up_floor(_, _, T), r_idle(T)), []).
rule(p_down_in_idle, prefer(r_down_in(_, _, T), r_idle(T)), []).
rule(p_down_floor_idle, prefer(r_down_floor(_, _, T), r_idle(T)),
[]).
rule(p_open_in_idle, prefer(r_open_in(_, T), r_idle(T)), []).
rule(p_open_floor_idle, prefer(r_open_floor(_, _, T),r_idle(T)), []).



% ====================================================================
%     Other Auxilliary Definitions
% ====================================================================

above(Floor1, Floor2) :-
      floor(Floor1), floor(Floor2), Floor1 > Floor2.

below(Floor1, Floor2) :-
      floor(Floor1), floor(Floor2), Floor1 < Floor2.

floor(Floor) :-
      member(Floor, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).


directly_above(Floor, Floor1) :-
      floor(Floor), floor(Floor1), Floor1 is Floor+1.

directly_below(Floor, Floor1) :-
      floor(Floor), floor(Floor1), Floor1 is Floor-1.


% ====================================================================
%     Main functions
% ====================================================================

% elevator(Example, Moves) :- Returns the set of moves the elevator
% should make for all time points, when actions of Example take place

elevator(Example, Moves) :-   example(Example),
                        elevator_moves(0, 9, Moves),
                        retractall(holds(_,_)),
                        retractall(rule(Example, _, _)),
                        retractall(rule(newaction, _, _)).


%elevator_moves(From, To, Actions) :- Returns the set of moves the
%elevator should  make between time points From and To in the list
%Actions

elevator_moves(Time, Time, [Action |[]]) :-
      !, give_action(Action, Time, _),
      write('Time '), write(Time), write(': '), write(Action),
write('\n').

elevator_moves(From, To, [Action |Rest]) :-
      give_action(Action, From, _),
      assert(rule(newaction, happens(action(Action, From), From),
[])),
      write('Time '), write(From), write(': '), write(Action),
write('\n'),
      Next is From + 1,
      elevator_moves(Next, To, Rest).




% assert_all(List) :- asserts all the rules in List after retracting
all instances of each item in order
% to make sure that each rule from list exists only once
assert_all([], _).
assert_all([First | Rest], T) :- retractall(holds(First, T)),
assert(holds(First, T)), assert_all(Rest, T).



% find_holds(T) :- finds every fact that holds at time T and asserts
a rule for each one
find_holds(T) :- findall(Fact, prove([holds(Fact, T)], _), List),
assert_all(List, T).

% give_action(Action, T, D) :- returns action Action that will be
executed at time T.
give_action(Action, T, D) :- find_holds(T), prove([action(Action,
T)], D).

% =====================================================================
%     Examples
% =====================================================================

example(example1) :-    assert(rule(example1,
holds(elevator_position(1), 0), [])),
                  assert(rule(example1,
happens(press_button_floor(1,up), 0), [])),
                  assert(rule(example1, happens(press_button_in(6),
1), [])),
                  assert(rule(example1, happens(press_button_floor(3,
up), 0), [])),
                  assert(rule(example1, happens(press_button_in(5),
4), [])).

example(example2) :-    assert(rule(example2,
holds(elevator_position(1), 0), [])),
                  assert(rule(example2,
happens(press_button_floor(1,up), 0), [])),
                  assert(rule(example2, happens(press_button_in(3),
1), [])),
                  assert(rule(example2, happens(press_button_floor(2,
down), 1), [])),
                  assert(rule(example2, happens(press_button_in(1),
6), [])).

% =====================================================================
%     Queries
% =====================================================================

% elevator(example1, Moves).
% elevator(example2, Moves).
%====================================================================
```