Using Event Calculus to Formalise Policy Specification and Analysis

Arosha K Bandara Emil C Lupu Alessandra Russo
Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
{bandara, e.c.lupu, ar3}@doc.ic.ac.uk

Abstract

As the interest in using policy-based approaches for systems management grows, it is becoming increasingly important to develop methods for performing analysis and refinement of policy specifications. Although this is an area that researchers have devoted some attention to, none of the proposed solutions address the issues of analysing specifications that combine authorisation and management policies; analysing policy specifications that contain constraints on the applicability of the policies; and performing a priori analysis of the specification that will both detect the presence of inconsistencies and explain the situations in which the conflict will occur.

This paper presents a method for transforming both policy and system behaviour specifications into a formal notation that is based on Event Calculus. Additionally it describes how this formalism can be used in conjunction with abductive reasoning techniques to perform a priori analysis of policy specifications for the various conflict types identified in the literature. Finally, it presents some initial thoughts on how this notation and analysis technique could be used to perform policy refinement.

1. Introduction

Policy based approaches to systems management are of particular importance because they allow the separation of the rules that govern the behaviour of a system from the functionality provided by that system [1]. This means that it is possible to adapt the behaviour of a system without the need to recode functionality, and changes can be applied without stopping the system. Research into policy based systems management has focussed on languages for specifying policies and architectures for managing and deploying policies in distributed environments. However, a much-neglected research problem is that of policy analysis and refinement. Whilst some previous work has investigated the nature of modality and application specific conflicts in policy specifications [2, 3], there still remain significant areas for improvement. In particular it is important to be able to analyse policies in the presence of constraints that control their applicability. When doing this, in addition to detecting the presence of conflicts, it is necessary to identify the exact causes for those conflicts to arise. Addressing these needs requires a formalism that will model both system behaviour and policy such that formal reasoning techniques can be used to analyse policy specifications.

The initial focus of our work has been to develop a formal representation for policies and the managed systems that will support formal reasoning techniques for detecting conflicts between policies. Existing policy specification formalisms [3-5] only handle security or management policies even though most real world systems use a combination of the two. Also, these approaches use deductive reasoning techniques for policy analysis, which require complete specification of the system state in order to produce useful results. Finally, it is important to be able to analyse a policy specification before it is deployed. However, none of the existing studies seem to model the system behaviour to support such a priori analysis of policy specifications that are constrained on the runtime state of the system.

In this paper, we propose a formalism that is based on the standard Event Calculus [6] to model both authorisation and management policy specifications together with system behaviour. Event Calculus was chosen as an appropriate basis for formalising policy specifications as both the policies and the management behaviour we are modelling are event driven. Additionally, since an Event Calculus specification of a system can be generated from a state transition model, users can specify the management behaviour using a familiar high-level notation. In a similar fashion, it is possible to translate policies specified in a high-level policy specification language, like Ponder [7], into an Event Calculus representation that describes the semantics of the policy language. This eliminates the need for the user to become conversant with the details of logic programming and the Event Calculus notation. Having developed the Event Calculus representation, we show how our formalism supports the specification of rules for detecting both modality conflicts and application specific conflicts, such as conflicts of duty. Using abductive reasoning techniques, we are able to analyse the policy specifications to identify existing conflicts and provide explanations on how they might arise. Because the abduction process is applied to a specification that models both the systems behaviour and the policy specification it is possible to detect conflicts when the applicability of the policies is constrained on the runtime state of the system. Furthermore, by using abduction, the analysis can be performed even with partial specifications of the system state. We also expect that this formalism will allow the use of other reasoning techniques likely to be useful in developing an approach for policy refinement.

Although the initial focus of our work has been on policy analysis, the end objective of this research is to develop tools and analysis techniques that will support policy refinement. Policy hierarchies and the application of policy refinement to derive lower-level, more specific policies from high-level ones are introduced in [8] and are motivated by the following needs:

- To determine the resources that are needed to satisfy the requirements of the policy.
- To translate high-level policies into operational policies that can be enforced by the system.
- To verify that the set of lower level policies actually meets the requirements of the high-level policy.

In requirements engineering, Darimont et al. present an approach that uses goal regression and refinement patterns that allow high-level requirements to be stated in terms of a combination of lower level ones [9]. In this paper, we follow a similar approach of developing policy refinement patterns, and outline a technique that uses abductive reasoning to ensure consistency and completeness when instantiating policies based on these patterns.

The next section presents a brief outline of the main policy types being considered, together with a description of the Event Calculus and the reasoning techniques being

```
// only a root process can cancel a print job
auth+      cancelJobRoot {
           subject    process/;
           target     printManager;
           action     cancelDoc(Job);
           when       process.owner == root;
}

// non root processes cannot cancel print jobs if
// the job being cancelled is not owned by the
// requesting process
auth-   cancelJobOther {
           subject   process/;
           target    printManager;
           action    cancelDoc(Job);
           when      process.owner != root &&
                  Job.owner != process.owner;
}
```

Figure 1: Examples of authorisation policies.

used in this work. Section 3 presents the complete specification language being used. Section 4 explains how the formalism can be used to detect various types of conflict. Section 5 outlines the initial approach for policy refinement. Sections 6 and 7 discuss the formalism presented and related work in this area of research. Finally, section 8 presents our conclusions together with some ideas for future work.

2. Background

2.1 Policy Specification

Existing research on policy based systems has identified several types of policy that are useful in managing distributed systems [7]. Broadly, policies can by classified into authorisation policies and management policies where the former category captures the access control requirements of a system and the latter category holds requirements related to the system behaviour. The Ponder language [7], developed at Imperial College, is a declarative language that supports both of these policy types.

Authorisation policies specify whether a subject is permitted perform a particular action on a target. In a closed system, with a default policy of prohibiting all subjects from performing operations on all targets, positive authorisation policies would be used to explicitly specify which particular operations a subject is permitted to perform on a target. Alternatively, in an open system, where any subject is allowed to perform any action on any target unless specifically excluded by a policy, negative authorisations would be used to specify that a subject is not permitted to perform an operation on a target. Examples of positive and negative authorisation as specified in the Ponder language are shown in Figure 1.

A policy-based access control system is the combination of the policies that specify the permitted/ prohibited operations, an access control model that defines how the permissions are organised across the system, and a reference monitor that uses the access control model to enforce the policies.

Obligation policies specify management operations that must be performed when a particular event occurs given some supplementary conditions being true. They are specified in terms of a subject that should perform a particular action on a target when a specified condition is true. Obligation policies are event based and therefore the occurrence of the specified event is a necessary condition for the mandated operation to be performed. Another difference is that obligation policies cause the agent enforcing the policy to actually perform the specified action rather than just specify that the operation is permitted. An example of an obligation policy is shown in Figure 2.

```
// Upon system shutdown, any jobs owned by running
// processes should be cancelled
oblig   shutdownCancellation {
          on        systemShutdown;
          subject   process/;
          target    printManager;
          action    cancelDoc(Job);
          when      Job.owner == process.owner;
}
```

Figure 2: Example of an obligation policy.

Refrain policies allow the administrator to specify conditions under which certain operations should not be performed. They are similar to negative authorisation policies as they are both used to prevent an action from being performed on a target. However, unlike authorisations, which are interpreted by the target object's access controller, refrain policies are interpreted by the subject and can be used in situations where the target does not wish to be protected from the subject such as information disclosure policies.

Prior work on policy specification has illustrated the power of using a domain model as a tool for organising objects in a system. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority. Membership of a domain is explicit and not defined in terms of a predicate on object attributes. An advantage of specifying policy scope in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies [10].

The Ponder language provides support for specifying authorisation, obligation and refrain policies. Its object oriented features and grouping constructs facilitate ease of use and scalability to large systems and large numbers of policies. However, Ponder is not a logic based language and does not provide direct support for formal reasoning methods or for expressing general models of system behaviour. Therefore, Ponder cannot account for the effect of policies on system state and cannot be used directly for policy analysis. However, as will be shown here, it is possible to transform Ponder policies into a formal representation that supports both a description of the system behaviour and formal reasoning techniques for policy analysis.

2.2 Event Calculus and Abductive Reasoning

Event calculus (EC) is a formal language for representing and reasoning about dynamic systems. Because the language supports a representation of time that is independent of any events that might occur in the system, it is a particularly useful way to specify a variety of event-driven systems. Since its initial presentation [6], a number of variations of the Event Calculus have been presented in the literature [11]. In this work we use the form presented in [12], consisting of (i) a set of time points (that can be mapped to the non-negative integers); (ii) a set of properties that can vary over the lifetime of the system, called *fluents*; and (iii) a set of event types. In addition the language includes a number of base predicates, `initiates`, `terminates`, `holdsAt`, `happens`, which are used to define some auxiliary predicates; and domain independent axioms. These are summarised below:

Base predicates:

initiates(A,B,T)	event A initiates fluent B for all time > T.
terminates(A,B,T)	event A terminates fluent B for all time > T.
happens(A,T)	event A happens at time point T
holdsAt(B,T)	fluent B holds at time point T. 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 sometime between timepoint T1 and T2.
declipped(T1,B,T2)	fluent B is initiated sometime between timepoint T1 and T2.

Domain independent axioms:

holdsAt(pos(B), T1) $\leftarrow$ initiallyTrue(B) $\wedge \neg$ clipped(T, B, T1) $\wedge$ T1=T+1.

holdsAt(pos(B), T1) $\leftarrow$ initiates(A, B, T) $\wedge$ happens(A, T) $\wedge \neg$ clipped(T, B, T1) $\wedge$ T1=T+1.

holdsAt(neg(B), T1) $\leftarrow$ initiallyFalse(B) $\wedge \neg$ declipped(T, B, T1) $\wedge$ T1=T+1.

holdsAt(neg(B), T1) $\leftarrow$ terminates(A, B, T) $\wedge$ happens(A, T) $\wedge \neg$ clipped(T, B, T1) $\wedge$ T1=T+1.

This is the classical form of the Event Calculus where theories are written using Horn clauses. The frame problem is solved by circumscription, which allows the completion of the predicates `initiates`, `terminates` and `happens`, leaving open the predicates `holdsAt`, `initiallyTrue` and `initiallyFalse`. This approach allows the representation of partial knowledge of the domain (e.g. the initial state of the system). Formulae derived by the Event Calculus are in effect classically derived from the circumscription of the EC representation. To provide an implementation of such a Calculus in Prolog, we use `pos` and `neg` functors. The semantics of the Prolog implementation assumes the Close Word Assumption (CWA) and models are essentially Herbrand models where predicates are appropriately completed. The use of `pos` and `neg` functions on the fluents allows us to keep open the interpretation of fluents being true/false, in the same way as circumscription does in the classical representation. In this way we can guarantee that the implementation of our EC is sound and complete with respect to the classical EC formalisation. The correspondence between the classical EC with circumscription and the logic program implementation can be found in [11].

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

The work described in [12] outlines how abduction can be used in conjunction with Event Calculus to analyse requirements specifications and presents a specialised set of Event Calculus axioms that reduce the computational complexity of the abductive proof procedure.

3. A Formal Language for Policies and Managed Systems

Because the enforcement of an obligation policy will change the state of the system, in addition to modelling the policy specification, it is necessary to model the system itself when developing a formal technique for analysing policies. To achieve a complete specification that supports formal reasoning, the following domain-specific information must be represented in the model.

- Objects and their organisation into domains.
- Available management operations and the effect they have on the managed objects.
- Policy rules.

Additionally, it is also necessary to define domain independent rules for modelling policy enforcement. In order to support the transformation of this information from high-level representations into a logical notation, we use the following constants, variables, functions and predicates:

1. **Constant Symbols:** Every member of `Obj`, where `Obj` represents the set of objects in the system.
2. **Variable Symbols:** These are defined using the set, V_O, representing the attributes of objects and V_P, representing the set of parameters for the operations supported by the objects.

Table 1: Function symbols.

Symbol	Description
state(Obj, V_O, Value)	Represents the value of a variable of an object in the system. It can be used in an `initiallyTrue` predicate to specify the initial state of the system and also as part of rules that define the effect of actions.
operation(Obj, Action(V_P))	Used to denote the operations specified in a policy function or event (see below)
systemEvent(Event)	Represents any event that is generated by the system at runtime and is used to trigger enforcement of obligation or refrain policies. The Event argument specified in this term can be any application specific predicate or function symbol.
doAction(Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)))	Represents the event of the action specified in the `operation` term being performed by the subject, Obj_{Subj}, on the target object, Obj_{Targ}.
requestAction(Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)))	Represents the event that occurs whenever a subject attempts to perform an operation on a target object. Therefore, this is the event that will trigger a permission (or denial) decision to be taken by the target object's access controller.
rejectAction(Obj_{Subj}, operation(Obj_{Targ},Action(V_P)))	Event that occurs after the enforcement decision to reject the request by a particular subject to perform an action is taken.
permit(Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))	Represents the permission granted to a subject, Obj_{Subj}, to perform the action defined in the operation on the target, Obj_{Targ}.
deny(Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))	Used to denote that the subject, Obj_{Subj}, is denied permission to perform that action on the target, Obj_{Targ}.
oblig(Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))	Denotes that the subject, Obj_{Subj}, should perform the action specified in the operation term on the target, Obj_{Targ}.
refrain(Obj_{Subj}, operation(Obj_{Targ}, Action (V_P)))	Denotes that the subject, Obj_{Subj}, should not perform the action specified in the operation term on the target, Obj_{Targ}.

Table 2: Predicate symbols.

Symbol	Description
object(Obj)	Used to specify that Obj is an object in the system.
attr(Obj, V_O)	Specifies that Vo is an attribute of the object, Obj.
method(Obj, Action(V_P))	Represents an action supported by an object in the system. It will be used to define a separate ground term for every operation specified in the system.
isDomain(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: object(Obj) ← isDomain(Obj).
isMember(Obj, Dom)	Holds if the object, Obj, is a member of the Domain, Dom.
isSubDomain(Dom1, Dom2) ← isDomain(Dom1), isDomain(Dom2), isMember(Dom1, Dom2), Dom1 != Dom2, ¬ isSubDomain(Dom2, Dom1).	Holds if the domain represented by Dom1 is a sub-domain of Dom2. The body of the rule is used to ensure that there are no cyclic relationships in the domain structure.
isDerivedMember(Obj, Dom) ← object(Obj), ¬ isDomain(Obj), isMember(Obj, Dom). isDerivedMember(Obj, Dom) ← object(Obj), ¬ isDomain(Obj), subDomain(Dom, SubDom), isDerivedMember(Obj, SubDom).	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.
isValidSpec(Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)) ← object(ObjSubj), object(ObjTarg), method(ObjTarg, Action(V_P)).	Many of the function definitions above contain the tuple (Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)). The isValidSpec predicate is defined to hold if the members of this tuple are consistent with the specification of the managed system. As such it is used in the body of any rule where functions with the tuple (Obj_{Subj}, operation(Obj_{Targ}, Action(V_P)) are specified in the head.

3. **Function Symbols:** The language supports a number of functions that can be used as parameters in the basic predicate symbols of Event Calculus (Table 1)

4. **Predicate Symbols:** In addition to the previously described Event Calculus predicates, initiates, terminates, happens, holdsAt and initiallyTrue, the language includes the predicate symbols defined in Table 2.

Having specified the language, it is now possible to explain how the various symbols defined above can be incorporated into rules that represent the different types of information required for modelling a managed system. The sequel presents the form of these rules and illustrates their use through a simple example.

3.1 Objects and Organisational Model

Consider an organisation that has a number of different printers distributed through its offices. The printers are organised according to properties like the type (colour/ b&w), capacity (high volume/ low volume) and physical location (4th floor/ 5th floor/ lab). The printers themselves are uniquely named (skyblue, violet, cobalt, grey, crimson, damson). A pictorial representation of the printer organisation is presented in Figure 3. Considering each of the properties to be represented by a different domain, the formalism presented above can be used to represent the printer crimson as an object in this domain structure.

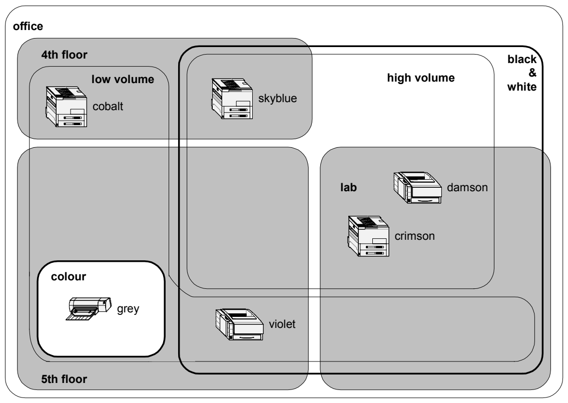


Figure 3: Domain structure for organisation of printers

```
object(printer-crimson).
attr(printer-crimson, status).
method(printer-crimson, printDoc).
method(printer-crimson, switchPaper).

isDomain(office).
isDomain(bw-printers).
isDomain(highvol-printers).
isDomain(lab).
isMember(lab, office).
isMember(highvol-printers, lab).
isMember(bw-printers, lab).
isMember(printer-crimson, lab).
isMember(printer-crimson, highvol-printers).
isMember(printer-crimson, bw-printers).
```

If the entire example is encoded in this manner, it is a simple matter to identify the set of printers that belong to any particular domain. For example, assuming that duplicate answers are removed, the following query will return all the printers in the office:

```
?- isDerivedMember(Printer, office).
   Printer = printer-skyblue; Printer = printer-violet ;
   Printer = printer-cobalt;  Printer = printer-grey ;
   Printer = printer-crimson; Printer = printer-damson;
```

3.2 System Behaviour Model

Having modelled the domain structure for organising the objects in the managed system, we now extend the language above, using Event Calculus, to model the operations supported by the system and their behaviour. The `method` symbol defined in Table 2 is used to represent the operations that are supported by the objects in the system. 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` and `terminates` predicates, which are defined in the Event Calculus, according to the following schema:

```
initiates(doAction(Obj_Subj, operation(Obj_Targ,
  Action(Parms))), PostTrue, Tm) ←
  validSpec(Obj_Subj, operation(Obj_Targ, Action(Parms)))
  ∧ PreCondition.

terminates(doAction(Obj_Subj, operation(Obj_Targ,
  Action(Parms))), PostFalse, Tm) ←
  validSpec(Obj_Subj, operation(Obj_Targ, Action(Parms)))
  ∧ PreCondition.
```

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`. In both of these rules, the `PreCondition` will be represented by a conjunction of `holdsAt` terms, which are defined as part of the Event Calculus. The `PostTrue` and `PostFalse` fluents are defined using `state` terms that are defined in the formal language above. The `validSpec` predicate is used to ensure that the objects and operations specified in the rule are consistent with the specification of the objects and their organisation e.g., the action specified is an operation defined in the interface of the managed object.

Building on the example used previously, it is possible to illustrate the use of these rules for modelling system behaviour. Consider that a print manager controls every printer in the system. The print manager provides functions for viewing the printer queue, adding and deleting a print job. Additionally it is possible for the printers to provide diagnostic information (such as a paper jam) to the print manager. The print manager can use the diagnostic information to correct errors, or report the printer status to a central management console that is monitored by an administrator. An UML state chart representation of this functionality is shown in Figure 4.

It is possible to transform this state chart into the Event Calculus notation presented previously where the input shown on each transition arrow is the action being performed; for transition between different states, the current state values become the `PostFalse` fluents; any actions associated with the transition and next state values become the `PostTrue` fluents; and the current state values become the `PreConditions`. Self-transitions should not specify the current state as `PostFalse` fluents. So following this scheme, transition (4) in Figure 4 would be represented in the Event Calculus as follows:

```
initiates(doAction(printer, operation(printer,
  switchPaper)), state(printer, status, busy), T) ←
  holdsAt(pos(state(printer, status, busy)), T) ∧
  holdsAt(pos(state(printMgr, status, jobSpooled)), T).
initiates(doAction(printer, operation(printer,
  switchPaper)),state(printMgr,status,jobSpooled), T) ←
  holdsAt(pos(state(printer, status, busy)), T) ∧
  holdsAt(pos(state(printMgr, status, jobSpooled)), T).
```

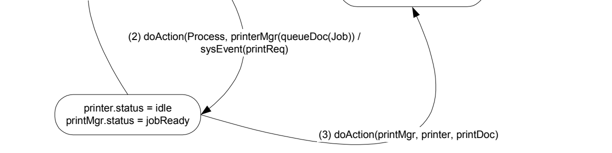

printer.status=idle
printMgr.status = waiting
(4) doAction(printer,printer, switchPaperTray)
printer.status = busy
printMgr.status = jobSpooled
Action(Process, printMgr, cancelDoc(Job))
(2) doAction(Process, printerMgr(queueDoc(Job)) / sysEvent(printReq)
printer.status = idle
printMgr.status = jobReady
(3) doAction(printMgr, printer, printDoc)

Figure 4: State chart for Printer system functionality.

3.3 Policy Enforcement Model

Analysis of policies requires the ability to determine the effect of a specified policy on the behaviour of the system. Therefore, in addition to modelling the policy specification, it is necessary to define rules that model the enforcement of the policies. Such rules have the effect of linking the policy specification to the system behaviour specification.

The complete policy enforcement model is illustrated in Figure 5. As shown, a system event is received by the subject's policy agent, which refers to the policy repository to determine if any of the obligation policies for this subject specify this event as a trigger. If there is an obligation, this will cause a request to perform the specified action to be sent to the target. If a refrain policy that prohibits this exists at the subject, then the action will be rejected. Once the subject makes a request to perform an action on the target, the target object's access controller processes it. To do this, the access controller evaluates the request by referring to the policy repository and the access control model of the system. If the action is permitted, the access control system will proceed to do the requested action. Otherwise, if the action should be denied, the access control system will reject the action.

The formal representation of this policy enforcement model presented is presented in Figure 6. The first rule models the behaviour of subject's policy agent, causing the event of requesting an action whenever an obligation that specifies that action holds. The next rule models a subject's policy enforcement code rejecting the specified action to enforce a refrain. The third rule models the behaviour of the target's access controller, generating a `doAction` event when an action is permitted. This event would trigger the relevant system behaviour rules thus causing the system state to change according to the specification. The last rule models a target object's access control monitor rejecting the action to prevent a denied operation from being performed.

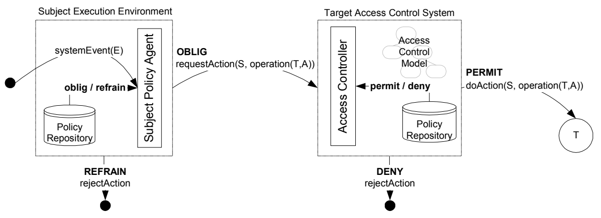


Figure 5: Policy enforcement model

```
% Obligation / Refrain Enforcement Rule (Subject)
happens(requestAction(Subj, operation(Targ,
 Action(ParmList))), Tn) ←
 holdsAt(oblig(Subj, operation(Targ, Action(ParmList))),Tm)
 ∧ (Tm < Tn).
happens(rejectAction(Subj, operation(Targ,
  Action(ParmList))), Tn) ←
  holdsAt(refrain(Subj, operation(Targ,Action(ParmList))),Tm)
  ∧ (Tm < Tn).

% Access Control Rule (Target)
happens(doAction(Subj, operation(Targ,
  Action(ParmList))), Tn) ←
  holdsAt(permit(Subj, operation(Targ, Action(ParmList))), Tm)
  ∧ (Tm < Tn).
happens(rejectAction(Subj, operation(Targ,
  Action(ParmList))), Tn) ←
  holdsAt(deny(Subj, operation(Targ, Action(ParmList))), Tm)
  ∧ (Tm < Tn).
```

Figure 6: Policy enforcement rules.

3.4 Policy Specification

The final step in developing this logical notation is to represent the policies themselves. As discussed in the previous sections, we are focussing on four types of policy – positive authorisation, negative authorisation, obligation and refrain.

In order to correctly interact with the enforcement model described above, each policy specification rule should initiate the appropriate policy function symbol (`permit`, `deny`, `oblig` or `refrain`) for each of the events. So for example, a positive authorisation policy rule should specify that `permit(Subj, Operation)` holds when the `requestAction(Subj, Operation)` event occurs and the constraints that control the applicability of the policy hold. Additionally, the fluent `permit(Subj, Operation)` should cease to hold once the action has been performed thus making it possible to re-evaluate the policy rule on subsequent requests to perform the action. The Event Calculus representation of this functionality is shown in the (posAuth) specification in Figure 7. This figure also shows how each of the other policy types would be represented by rules in the formal notation.

For each rule, the terms, $\texttt{Obj}_{\texttt{Subj}}$, $\texttt{Obj}_{\texttt{Targ}}$, `Action` and `Constraint`, can be directly mapped to the subject, target, action, constraint and event clauses used when specifying policies in a language like Ponder. Although Ponder constraints are specified using the Object Constraint Language (OCL), typical constraints only use a subset of features from this language. As such, the `Constraint` predicates in the Event Calculus rules above, can be represented by a combination of `holdsAt` terms. Beckert et al. [13] describe approaches for mapping general OCL specifications into first order logic. This could be used to handle more complex OCL constraint expressions. The `validSpec` predicate, which is the second predicate in the body of these policy rules, is used to check that the objects and operations specified in the rules are consistent with the system description.

The (negAuth) rule in Figure 7 represents a negative authorisation policy by stating that, if the `Constraint` holds and the event requesting the action is performed happens, the action is denied. The second part of the (negAuth) rule shows how the `deny` fluent will be terminated once the decision to reject that action has been taken, thus allowing the rule to be re-evaluated on subsequent requests. Note that the termination rules for these policies do not have any constraints and can be generically specified for the whole system.

The (oblig) rule states that if the `Constraint` holds at the time that the system event, `systemEvent(E)`, occurs, then the obligation for the subject to perform the action on the target holds. Like with the (posAuth) rule, we define that the obligation is terminated once the call to perform the specified operation is made. This assumes that the execution of the operation is an atomic process,

```
(posAuth) - initiates(requestAction(ObjSubj,operation(ObjTarg, Action(ParmList))),permit(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))) ∧ Constraint.

      terminates(doAction(ObjSubj, operation(ObjTarg, Action(ParmList))), permit(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))).

(negAuth) - initiates(requestAction(ObjSubj,operation(ObjTarg, Action(ParmList))),deny(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm)   ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))) ∧ Constraint.

      terminates(rejectAction(ObjSubj, operation(ObjTarg, Action(ParmList))), deny(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))).

(oblig)   - initiates(systemEvent(E), oblig(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))) ∧ Constraint.

      terminates(doAction(ObjSubj, operation(ObjTarg, Action(ParmList))), oblig(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))).

(refrain) - initiates(systemEvent(_),refrain(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm)   ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))) ∧ Constraint.

      terminates(rejectAction(ObjSubj, operation(ObjTarg, Action(ParmList))), oblig(ObjSubj, operation(ObjTarg, Action(ParmList))), Tm) ←
                        validSpec(ObjSubj, operation(ObjTarg, Action(ParmList))).
```

Figure 7: Event Calculus representation of policies.

i.e. the execution of the operation is considered complete once a call to the operation has been made. The (refrain) rule states that if the Constraint holds and any system event occurs, the subject should not perform the action on the target because the refrain holds. Just like with the (negAuth) rule, the second part of the (refrain) rule defines that the refrain fluent is terminated once the policy enforcement decision to not perform the specified action is taken.

A complete policy specification would involve instantiating the initiates rules defined above with specific subjects, targets and operations defined for the managed system. The rules simply define the conditions under which a policy holds in the system.

The use of the policy specification rules defined previously can be illustrated by extending the printer management example to include a range of policy rules. Policies could be used to specify the types of process that are allowed to access the print queue. For example, only root processes are allowed to indiscriminately delete jobs from a queue. A user process is only allowed to delete a print job if it has the same process identifier as the process that originated the job. The print manager should handle an outOfPaper event by switching to an alternative input tray also reporting the event to the central console.

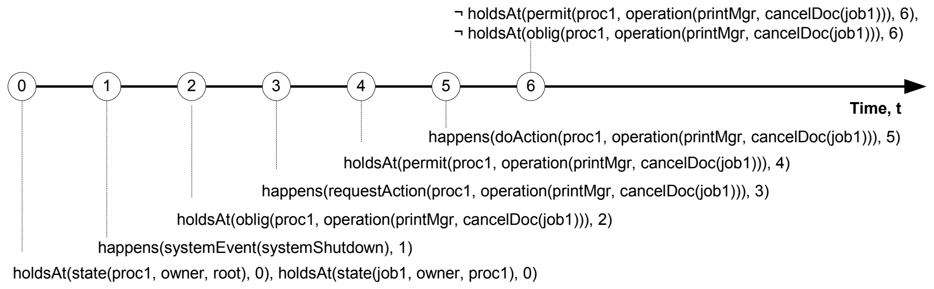
We can use the notation described in this section to represent the Ponder policies shown in Figures 1 & 2 as follows:

```
% Authorisation
initiates(requestAction(Process, operation(printMgr,
 cancelDoc(Job))), permit(Process, operation(printMgr,
 cancelDoc(Job))), T) ←
 validSpec(Process, operation(printMgr, cancelDoc(Job)))
 ∧ holdsAt(pos(state(Process, owner, root)), T).

initiates(requestAction(Process, operation(printMgr,
 cancelDoc(Job))), deny(Process, operation(printMgr,
 cancelDoc(Job))), T) ←
 validSpec(Process, operation(printMgr, cancelDoc(Job)))
 ∧ holdsAt(neg(state(Process, owner, root)), T)
 ∧ holdsAt(neg(state(Job, owner, Process)), T).

% Obligation
initiates(systemEvent(systemShutdown), oblig(Process,
 operation(printMgr, cancelDoc(Job))), T) ←
 validSpec(Process, operation(printMgr, cancelDoc(Job)))
 ∧ holdsAt(pos(state(Job, owner, Process)), T).
```

The interaction between these policy rules and the enforcement, and behaviour model is illustrated in Figure 8. Here, the initial system state consists of a process, proc1, owned by ‘root’ and a print job, job1 that is owned by that process. When the systemShutdown event occurs at t=1, this triggers the obligation rule shown above. The assertion of the obligation fulfils the condition of the obligation enforcement rule and causes a request to perform the cancelDoc(Job) action to be generated. This event triggers the first authorisation policy rule above, causing the operation to be permitted, which then satisfies the condition of the access control enforcement rule.



¬ holdsAt(permit(proc1, operation(printMgr, cancelDoc(job1))), 6),
¬ holdsAt(oblig(proc1, operation(printMgr, cancelDoc(job1))), 6)
0
1
2
3
4
5
6
Time, t
happens(doAction(proc1, operation(printMgr, cancelDoc(job1))), 5)
holdsAt(permit(proc1, operation(printMgr, cancelDoc(job1))), 4)
happens(requestAction(proc1, operation(printMgr, cancelDoc(job1))), 3)
holdsAt(oblig(proc1, operation(printMgr, cancelDoc(job1))), 2)
happens(systemEvent(systemShutdown), 1)
holdsAt(state(proc1, owner, root), 0), holdsAt(state(job1, owner, proc1), 0)

Figure 8:Timeline of interactions between policy rules and enforcement model

After the `doAction(…)` event occurs at t=5, the termination rules specified in the enforcement model cause the `permit` and `oblig` fluents to be terminated at t=6.

4. Policy Analysis

Since the policy specification notation described above supports policy types that are semantically opposite to each other (e.g. obligations and refrains), it is to be expected that conflicting policy specifications could arise. Therefore, it is important to provide a means of detecting conflicts in the policy specification as part of the logical framework.

Several different types of conflicts that can occur in a policy specification are identified in [2]. Modality conflicts are those that arise when two policies are specified using the same subjects, targets and actions but are of opposite modality (e.g. positive and negative authorisations). This type of conflict can be considered to be domain-independent since conflicts could occur irrespective of the application domain for which the policies are being specified. Other types of conflict identified in the literature fall into the category of application specific conflicts. As described in [14], these include conflicts of duty, conflicts of interest, multiple manager conflicts, conflicts of priorities for resources and self-management conflicts.

Considering the types of conflict described above, it is possible to define rules that can be used to recognise conflicting situations in the policy specification.

4.1 Modality Conflicts

Modality conflicts involving authorisation policies occur when there are two policies, one an authorisation and the other a prohibition, defined for the same subject, target and action. The `authConflict` predicate defined below holds if an authorisation conflict is detected.

```
holdsAt(authConflict(Subj, Op), Tm) ←
 holdsAt(permit(Subj, Op), Tm) ∧  holdsAt(deny(Subj, Op), Tm).
```

In a similar fashion, rules for detecting conflicts between obligations and refrains; and unauthorised obligation conflicts can be defined as follows:

```
holdsAt(obligConflict(Subj, Op), Tm) ←
  holdsAt(oblig(Subj, Op), Tm) ∧ holdsAt(refrain(Subj, Op), Tm).

holdsAt(unauthObligConflict(Subj, Op), Tm) ←
  holdsAt(oblig(Subj, Op), Tm) ∧ holdsAt(deny(Subj, Op), Tm).
```

In each of these rules, the `Op` variable will be instantiated with an `operation` term as defined in Section 3.

4.2 Application Specific Conflicts

One of the most common types of application specific conflict cited in the literature is conflict of duties (alternatively stated as the requirement to ensure separation of duties) [3, 14-16]. A conflict of duties will arise if the same subject is permitted to perform operations that, in the context of the application, are defined to be conflicting. For example, in a company financial system, the operation of entering a request for payment and the operation of approving that request are potentially conflicting if the same user can perform both operations.

Rules for application specific conflicts must be defined using constraints that include application specific data in addition to policy information. However, before defining rules for detecting such conflicts, it is important to have a means of specifying this application specific information. The description of the various types of application specific conflict in [14], suggests that the application specific information required to detect these conflicts is a definition of operations that are incompatible. Two operations are considered incompatible with each other if:

- A conflict of duty arises when the same subject performs both operations on the same target (e.g. an employee makes a payment request and approves it).
- A conflict of interest arises when the same subject performs each of the operations on different targets. (e.g. a bank provides investment advice to a client whilst performing a merger for a competing client).
- Different subjects perform each of the operations on a single target and the outcome of each operation is incongruent with the other. (e.g. spooling a job to a printer and shutting the same printer down).

In order to capture this application specific information, we extend the system specification language described in Section 3 with a new symbol – `conflictingOps(ConflictType, [Ops])`. Here the `ConflictType` represents a constant value from the set {`conflictDuty`, `conflictInterest`, `conflictGoal`, `conflictSelfMgmt`}, indicating the type of application specific conflict that may arise if the operations are used in a policy specification. The members of the `Ops` list are instantiated using the `operation` term defined previously. The symbol can be used to define ground literals in the system specification, specifying the action/target object combinations that will potentially conflict. In the case of the conflict of duties example mentioned above, the potential conflict between the operations of requesting a payment and approving a payment would be represented as follows:

```
conflictingOps(conflictDuty, [operation(payment,
request(PaymentID,Amount)),operation(payment,approve(PaymentID))])
```

As described in the literature, the principle of separation of duty can take a number of different forms. In the first case, static separation of duty is ensured by not permitting a subject to perform an operation, `Op1`, if that subject has ever been granted permission for a different operation, `Op2`, and `Op1` and `Op2` are defined as members of a set of conflicting operations. A policy specification that violates this principle will give rise to a conflict of duty. The second variation, dynamic separation of duty, requires that the runtime behaviour of the system should not allow conflicting operations to be performed. Finally, [15] presents a specialised form of dynamic separation of duty called the Chinese Wall policy that is intended to prevent a subject performing any conflicting actions on one target, if the subject has already been given permission to perform a conflicting action on a different target. A comprehensive formal treatment of separation of duty policies is presented in [16].

In the formalism presented here, we model the dynamic behaviour of the system because this is necessary for dealing with the effects of having constraints in the policy specification. This allows us to treat the detection of static and dynamic conflicts of duty in a similar manner by defining rules of the following form, depending on the number of operations that could cause conflicts:

```
holdsAt(sepOfDutyConflict(Subj, Ops), Tm) ←
holdsAt(permit(Subj, Op1), T1) ∧ holdsAt(permit(Subj, Op2), T2) ∧
... ∧ holdsAt(permit(Subj, OpN), TN) ∧
conflictingOps(conflictDuty, Ops) ∧ memberOf(Op1, Ops) ∧
memberOf(Op2, Ops) ∧ ... memberOf(OpN, Ops) ∧ T1=<T2=<...=<TN=<Tm.
```

The rule for detecting a conflict in a Chinese Wall policy is different because the conflict condition also depends on the targets involved. We represent this as follows:

```
holdsAt(cwConflict(Subj,Target1,Action1,Target2,Action2), Tm) ←
holdsAt(permit(Subj, operation(Target1, Action1)), T1) ∧
holdsAt(permit(Subj, operation(Target2, Action2)), T2) ∧
conflictingOps(conflictDuty, Ops) ∧ Target1 != Target2 ∧
memberOf(operation(Target1, Action1), Ops) ∧
memberOf(operation(Target2, Action2), Ops) ∧
T1 =< T2 =< Tm.
```

Another type of conflict, identified in the literature as a multiple management conflict, arises when different subjects attempt to perform actions on the same target, where the goals of those actions are incongruent. For example, spooling a job to a printer and shutting the same printer down are operations with incompatible goals. We represent these operations using the constant, `conflictGoal`, in the `conflictingOps` term. The following is a representation of the printer example above using this symbol:

```
conflictOfGoalsOps(conflictGoal, [operation(printer,
  printDoc), operation(printer, shutDown)]).
```

Once the incompatible operations have been defined, the following rules can be used to identify multiple manager conflicts in a policy specification:

```
holdsAt(conflictOfMultiManagers(Subj1, Subj2, ..., SubjN
                                Ops), Tm) ←
holdsAt(permit(Subj1, Op1), T1) ∧
holdsAt(permit(Subj2, Op2), T2) ∧  ...  holdsAt(permit(SubjN,
OpN), TN) ∧
conflictingOps(conflictGoal, Ops) ∧
memberOf(Op1, Ops) ∧
memberOf(Op2, Ops) ∧  ...
memberOf(OpN, Ops) ∧ T1 =< T2 =< ... =< TN =< Tm.
```

Similar rules are specified for other types of application specific conflicts, such as conflicts of interest and self-management conflicts.

4.3 Detecting Conflicts

By using one of the conflict fluents (e.g. `unauthObligConflict`) as a goal state, it is possible to query the system specification for event sequences that would result in a conflict occurring. If no such sequence can be derived, it can be considered that the policy specification is free of this particular conflict type.

The current implementation of the analysis system makes use of the abductive proof procedure presented in [12]. By treating the conflicts fluents as safety properties of the system, this technique reduces the complexity of the abductive proof procedure to two time points – the time before the conflict arises (`t`) and the time after it arises (`t1`). Additionally, provided the conflict term is specified using ground literals, it can be shown that the query will always generate a complete explanation for any conflicts and it will always terminate [12].

Figure 9 shows an illustration of performing such a query on the example system presented previously. Here some of the solutions, such as the last, present the trivial case in which a conflict might occur. However, the first solution suggests that there is a sequence of events that will cause a conflict even when there is no obligation or refrain that holds in the initial state. Therefore, it can be concluded that the policy specification contains a conflict.

```
?- demo([holdsAt(obligConflict(printMgr,
   operation(printer, printDoc)), t1)], [], Plan).
Plan = [initiallyTrue(state(printMgr,state,
  shuttingDown)), happens(systemEvent(printReq),t)]
Plan =  [initiallyTrue(refrain(printMgr,
  operation(printer,printDoc))),
happens(systemEvent(printReq),t)]
Plan = [happens(systemEvent(printReq),t),
  initiallyTrue(state(printMgr,state,shuttingDown)),
  initiallyTrue(oblig(printMgr,operation(printer, printDoc)))]
Plan = [initiallyTrue(refrain(printMgr,
   operation(printer,printDoc))),
   initiallyTrue(oblig(printMgr, operation(printer,printDoc)))]
```

Figure 9: Example of a conflict detection query

5. Policy Refinement

The ability to specify policies and managed system behaviour in a notation that supports formal analysis that allows detection of inconsistencies is a worthy goal in its own right. However, the longer term motivation for the work presented here, and a key problem area that we seeking to address is the development of new techniques for policy refinement. In this section we present our initial ideas for a technique that makes use of the formalism presented above.

The objective of policy refinement is to transform high-level policy specifications into more specific policies that are defined in terms of lower-level entities and operations of the system. More formally policy refinement could be defined as follows:

Definition: ***(Policy Refinement)*** If there exists a set of policies Prs:p1, p2, .. pn, such that the enforcement of a combination of these policies results in a system behaving in an identical manner to a system that is enforcing some base policy Pb, it can be said that Prs is a refinement of Pb. The set of policies Prs:p1, p2, .. pn is referred to as the refined policy set.

Much of the work done in the requirements engineering domain for refining goals into implementation specifications could be applicable to policy refinement. Using this definition and drawing on work done to identify the properties of goal refinements [9] the following properties are proposed:

1. **Correctness:** a refinement is said to be correct if the conjunction of all the members of that subset is a refinement of the base policy.
2. **Consistency:** a refinement is said to be consistent if there are no conflicts between any of the policies in the refined policy set.
3. **Minimality:** a refinement is said to be minimal if it is correct and if removing any policy from the refined policy set causes the refinement to be incorrect.

In addition, a policy refinement can be said to be complete iff all the properties defined above hold. The goal refinement approach also specifies a fourth property, non-triviality, which requires there to be more than one element in the refined set. However, in the policy refinement domain it may be acceptable to have a single policy that is a refinement of some base policy, provided that the refinement uses subjects, targets and actions that map to different physical entities. Therefore we do not consider this property to be a requirement of completeness in a policy refinement.

So, an essential requirement when refining a policy is to ensure that the goal achieved by that policy would still be achieved by the set of sub-policies that it is refined into. Having a formalism that allows abductive reasoning offers some useful capabilities in this regard since such formalisms support goal regression. Goal regression is a logical analysis technique that derives plans of action for achieving a specified end goal [17]. The desired end goal will be determined by the post-conditions of the operation specified in the base policy to be refined and abductive goal regression can be applied to derive the set of subject/operation tuples (of the form `[(Subj1, operation(Targ1, Action1)), … (SubjN, operation(TargN, ActionN))]`) that will be used by the refined policy set. Because this procedure is based on a formal proof procedure, the derived set of subjects and operations will be correct and minimal.

Having derived the set of subjects/operation combinations that will achieve the end goal of the base policy, it is now necessary to compose them into a refined policy set. The manner in which this composition procedure is performed is dependant on the type of the base policy and any application specific constraints that need to be applied. For example, if the goal of the operation specified in an authorisation policy can be refined to the set of subject/operation tuples, `[(Subj1, operation(Targ1, Action1)), … (SubjN, operation(TargN, ActionN))]`, one possibility for refinement of the original policy is to create a new authorisation policy for each of these subject/operation tuples and use the analysis techniques described here to validate that these new policies do not lead to any inconsistencies. Alternatively, it may be necessary to limit the number of new authorisations created to ensure that some application-specific constraint, such as a principle of least privilege, is observed. This would require an alternative scheme for composing the subject/operation tuples into refined policies. Each of these composition schemes can be considered to be a *refinement pattern* that is parameterised by factors such as the base policy type, the types on conflicts that should be checked for, any additional application specific constraints etc. Certain patterns will apply to certain types of base policies, whereas others might be generically applicable to any base policy. By developing a library of such patterns, it should be possible for the administrator to select any policy in the system and perform a refinement by applying a valid refinement pattern. However, significant work remains to be done towards this goal.

6. Discussion

When developing a formal language of the type presented here, it is important that it is expressive enough to represent the systems being modelled and that the language is based on solid theoretical foundations. Event calculus is a good starting point for a formal language for specifying policy-based systems because it has direct support for representing the events used in these systems. Additionally, it is a well-researched area of logic programming that supports all modes of logical reasoning and provides a number of theoretical results and tools that have been leveraged in this work. For example, the use of event calculus allows users to specify the system behaviour using more familiar notations, such as state charts, which can then be automatically translated into the logic program representation. Through the example presented here, it has been shown that the language is sufficiently expressive to model system objects, their organisation and behaviour together with policy rules that specify authorisations, obligations and refrains. Also, the language supports analysis of the specification for detecting both modality conflicts and a range of application specific conflicts. A particular strength of the abductive analysis technique presented here is its ability to perform a priori analysis of partial specifications and not only detect the existence of potential conflicts but also to generate explanations for the conditions under which conflicts may arise.

Another important consideration in any formal technique is the decidability and computational complexity of the algorithm used. We have briefly mentioned that using ground literals in any query term ensures termination of the conflict search process. Additionally, the formalism presented limits its use of first-order logic to stratified logic [18]. This permits a constrained use of recursion and negation while disallowing those combinations that lead to undecidable programs. Indeed, there are numerous studies that identify stratified logic as a class of first order logic that supports logic programs that are decidable [19, 20]. Moreover, such programs are decidable in polynomial time [3]. A more detailed analysis of the computational complexity and expressive power of stratified logic can be found in [20].

Although the formalism presented models different policy types and supports analysis for detecting a range of inconsistencies, there exist certain limitations to its capabilities. For example, there is no support for grouping policies into structures such as roles and other management grouping described in [7]. Additionally, the formalism does not model meta-policies and the interaction of these policies with the underlying enforcement architecture. It would be necessary to extend the language to support these constructs. Finally, the current abductive proof procedure only provides basic diagnostic information about the event history that leads to inconsistencies. This is because we reduce the problem to reasoning over just two time points. This limitation can be addressed by using a more powerful tool, such as the A-system [21], which has the capability to perform abductive reasoning over an arbitrary time line.

7. Related Work

Amongst the many alternative approaches to policy specification, there are a number of proposals for formal, logic-based notations. In particular Logic-based languages have proved attractive for the specification of security policy, as they have a well-understood formalism, which is amenable to analysis. However they can be difficult to use and are not always directly translatable into efficient implementation. A number of formalisms for security policy assume a role based access control (RBAC) model, including RSL99[22], Role Definition Language [23] and Temporal RBAC (TRBAC) [24]. Additionally there are languages that take advantage of the computational efficiencies offered by using subsets of first order logic, such as stratified logic. In [25], Barker presents a language that supports specification of access control policies using stratified clause-form logic, with emphasis on RBAC policies. However, this work does not discuss techniques for detecting conflicts in policy specifications. The Authorisation Specification Language (ASL) proposed by Jajodia et al. [3] is another example of a language based on stratified clause-form logic that also offers techniques for detecting modality conflicts and some application specific conflicts in authorisation policy specifications. However, this technique does not support static analysis of policy specifications that use constraints, assuming instead that conflict detection will take place at runtime.

The Policy Description Language (PDL) [4] is an example of first-order logic being applied to the specification of obligation policies. The language can be described as a real-time specialised production rule system to define policies. The syntax of PDL is simple and policies are described by a collection of two types of expressions: *policy rules* and *policy defined event propositions*. Later work by Chomicki [5], extends PDL to include the concept of action constrains, which are policies that prevent a specified action from being performed in a given situation. These action constraints are analogous to the refrain policies described in this paper. This work introduces the idea of using a policy monitor to detect conflict situations and resolve them by either suppressing the events that could lead to a conflict or overriding the conflicting action. Additionally, work by Son and Lobo, presents an approach for reasoning about policies with the objective of mapping a desired action history back to a possible event history [26]. This work is interesting because it illustrates how formal techniques together with logic programming can be used to derive information about the policy program – in this case the event history that causes a particular set of actions. However, PDL does not model authorisation policies and therefore the analysis cannot detect conflicts involving authorisations.

Recent work on using policies for adaptation of mobile devices [27] proposes Event Calculus as a suitable formalism for policy specification. However, this technique only models obligation policies and support for conflict detection using the notation is still under development. Finally there is ongoing work at Imperial College to develop a formal language for contract representation that is using Event Calculus as a baseline notation. It is expected that the notation presented here would be of particular relevance to this effort.

There are few examples of practical approaches for policy refinement. One such example is described in work done at Hewlett-Packard Laboratories, which outlines a policy-authoring environment that provides a policy wizard tool, called POWER, for refining policies [28]. Here, a domain expert first develops a set of policy templates, expressed as Prolog programs, and the policy authoring tools have an integrated inference engine that interprets these programs to guide the user through the refinement process. A major limitation of this approach is the absence of any analysis capabilities to evaluate the consistency of the refined policies. Also, the POWER approach depends on the domain expert having a detailed understanding of the entire system to develop a usable policy template. The refinement approach outlined in this paper avoids these problems by not only incorporating a complete analysis technique but also supporting abductive reasoning for deriving the action sequences required to achieve a goal.

Work done at University College London proposes using model checking for verifying the consistency of rules specified for a DiffServ router [29]. This technique depends on generating packet flows that can be used by the model-checking tool. However it is not possible to generate a complete set of packets that would ensure an exhaustive verification of the specification. Additionally, many of the packets generated will be benign – causing no inconsistencies in the system. By modelling the DiffServ modules using the formalism presented in this paper, it would be possible to use abduction to derive just those packets that could cause an inconsistency to arise. We are currently in the process of coding an example DiffServ router configuration and associated rules to validate this approach.

8. Conclusions and Future Work

In this paper we have described the use of Event Calculus and abductive reasoning for developing a language that supports specification and analysis of policy based systems. The language is sufficiently expressive to model systems using a combination of authorisation, obligation and refrain policies. Additionally we have shown how an abductive analysis procedure can be used to detect modality conflicts and a range of application specific conflicts.

We outline an initial approach for using the formalism presented for refining policies. Developing a library of refinement patterns that can be used in conjunction with the abductive analysis technique presented here will be the focus of our future work. Also, as part of this work we will look at developing tools to support the specification, analysis and refinement of policies using this formal notation. Additionally we are hoping to apply this formalism to a network management example that uses policy-based management for QoS provision in DiffServ networks.

Acknowledgements

We acknowledge financial support for this work from the EPSRC (Grant No: GR/R31409/01) and CISCO Systems Inc. Additionally, we would like to thank Morris Sloman and Naranker Dulay for their valuable feedback during the preparation of this paper.

References

[1] M. S. Sloman, "Policy Driven Management for Distributed Systems," *Journal of network and Systems Management*, vol. 2, pp. 333-360, 1994.
[2] E. C. Lupu and M. S. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *In IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management*, vol. 25, pp. 852-869, 1999.
[3] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A Logical Language for Expressing Authorisations," presented at IEEE Symposium on Security and Privacy, Oakland, USA, 1997a.
[4] J. Lobo, R. Bhatia, and S. Naqvi, "A Policy Description Language," presented at 16th National Conf. on Artificial Intelligence, Orlando, Florida, USA, 1999.

[5] J. Chomicki, J. Lobo, and S. Naqvi, "A Logic Programming Approach to Conflict Resolution in Policy Management," presented at 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA, 2000.

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

[7] N. Damianou, N. Dulay, E. C. Lupu, and M. S. Sloman, "The Ponder Policy Specification Language," presented at Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK, 2001.

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

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

[10] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman, "Tools for Domain-based Policy Management of Distributed Systems," presented at Network Operations and Management Symposium (NOMS 2002), Frorence, Italy, 2002a.

[11] R. Miller and M. Shanahan, *The Event Calculus in Classical Logic Alternative Axiomatisations*, vol. 4. Linköping , Sweden: Linköping University Electronic Press - http://www.ep.liu.se/ea/cis/1999/016/, 1999.

[12] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer, "An Abductive Approach for Analysing Event-Based Requirements Specifications," presented at 18th Int. Conf. on Logic Programming (ICLP), Copenhagen, Denmark, 2002.

[13] B. Beckert, U. Keller, and P. H. Schmitt, "Translating the Object Constraint Language into First-order Predicate Logic," presented at VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark, 2002.

[14] J. D. Moffett and M. S. Sloman, "Policy Conflict Analysis in Distributed System Management," *Journal of Organisational Computing*, vol. 4, pp. 1-22, 1994.

[15] D. F. C. Brewer and M. J. Nash, "The Chinese Wall Security Policy," presented at IEEE Symposium on Research in Security and Privacy, Oakland, California, USA, 1989.

[16] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo, "On the Formal Definition fo Searation-of-Duty Policies and their Composition," presented at IEEE Symposium on Security and Privacy, Oakland, CA, 1998.

[17] J. L. Pollock, "The Logical Foundations of Goal-Regression Planning in Autonomous Agents," *Journal of Artificial Intelligence*, vol. 106, 1998.

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

[19] G. Jager and R. F. Stark, "The Defining Power of Stratified and Hierarchical Logic Programs," *Journal of Logic Programming*, vol. 15, pp. 55-77, 1993.

[20] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and Expressive Power of Logic Programming," presented at 12th Annual IEEE Conf. on Computational Complexity (CCC'97), Ulm, Germany, 1997.

[21] B. van Nuffelen and A. Kakas, "A-System : Programming with abduction," presented at Logic Programming and Nonmonotonic Reasoning (LPNMR 2001), 2001.

[22] G.-J. Ahn and R. Sandhu, "The RSL99 Language for Role-based Separation of Duty Constraints," presented at Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1999.

[23] R. J. Hayton, J. M. Bacon, and K. Moody, "Access Control in an Open Distributed Environment," presented at IEEE Symposium on Security and Privacy, Oakland, California, U.S.A., 1998.

[24] E. Bertino, P. Bonatti, and E. Ferrari, "TRBAC: A Temporal Role-Based Access Control Model," presented at 5th ACM Workshop of Role-Based Access Control, Berlin, Germany, 2000.

[25] S. Barker, "Access Control Policies as Logic Programs," Imperial College of Science, Technology and Medicine, London, MPhil/PhD Transfer Report December 2001.

[26] T. C. Son and J. Lobo, "Reasoning about Policies Using Logic Programs," presented at AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, Stanford University, CA, 2001.

[27] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst, "Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems," presented at Third Int. Workshop on Policies for Distributed Systems and Networks (POLICY-2002), Monterey, CA, USA, 2002.

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

[29] L. Zanolin, C. Mascolo, and W. Emmerich, "Model Checking Programmable Router Configurations," University College London, London UK, Research Note Nov 2002.