

# Dynamic Conflict Detection in Policy-Based Management Systems

Nicole Dunlop †§, Jadwiga Indulska †§, Kerry Raymond §

† School of Information Technology and Electrical Engineering,  
The University of Queensland, Australia

§ CRC for Enterprise Distributed Systems Technology (DSTC), Australia

E-mail: {dunlop, jaga}@itee.uq.edu.au, kerry@dstc.edu.au

## Abstract

*While advances in open distributed systems have undoubtedly provided a uniquely diverse environment for users, managing the resources within such an environment has become an increasingly complex task. This challenge has been considered for several years within the distributed systems management research community and we have recently seen policy-based management emerge as one such promising exemplification.*

*The focus of our work has been predominantly on supporting the requirements of large evolving enterprises. Such environments present a significant challenge for policy-based management as the fluidity and complexity of interactions occurring in such environments mean that prevailing static-based specification and analysis of policies and roles, would be inadequate in many instances. We are therefore interested in providing support for a dynamic policy-based management environment.*

*This paper discusses the critical nature of providing both dynamic and static conflict detection and resolution and introduces a scalable computationally-efficient dynamic conflict detection mechanism.*

## 1 Introduction and Motivation

It is well understood that the increasing importance of open distributed systems and the growing number of services that utilise them has given rise to the need for effective distributed systems management [17]. Recently, policy-based management has been gaining wide acceptance as a means of enforcing enterprise-specific procedures; however, the specific nature of the requirements of large evolving enterprises have been predominantly unconsidered.

In order to support large evolving enterprise it is typically necessary to manage a large set of diverse objects,

across diverse organisational boundaries, over a potentially extensive lifetime. Users need to be able to adapt to changing circumstances and organisational responsibilities and to make use of new services as they are introduced.

The nature of managing large sets of organisational objects across diverse boundaries is that conflicting requirements often emerge, which subsequently materialise into policy specification conflicts. Clearly there is limited value in developing policy-based management models that do not provide ensuing support for conflict detection and resolution. While a significant attempt at *static* conflict detection has been made by Imperial College [8], the very crucial and complex issue of dynamic conflict detection in policy-based management has gone largely unresolved. Furthermore, our research has revealed that there exists a large class of policy-based conflict which simply cannot be determined statically.

We believe there exist two broad classes of conflict that need to be understood and independently managed; these are static and dynamic conflicts. It is important to make the distinction between these two classes of conflict as detection and resolution can be computationally intensive, time-consuming and hence, expensive and is most *preferably* done statically, at compile-time. However, dynamic or potential conflict is quite unpredictable, in that it may, or may not, proceed to a state of realised conflict; that is, the inconsistency may be exposed temporarily, or indeed not at all. This class of conflict must be detected at run-time.

Being able to detect conflict both statically and at run-time relies on a knowledge of the temporal characteristics of the policy as policy conflict will only occur when the objectives of two or more active policies cannot be simultaneously met. Through careful specification of policy types through combining both the deontic and temporal properties of policy, we have been able to classify conflict and develop a set of conflict profiles that assist us in determining the nature of both static and dynamic conflict within a policy specification. We have also developed a series of algorithms for conflict detection and implemented these within a proto-

type which demonstrates that it is possible to detect conflict at run-time in a way that is computationally-efficient.

While some work on detecting dynamic conflict has also been initiated in the area of telecommunications feature interaction [18, 20], it is not particularly applicable to reasoning about the special nature of conflicts surrounding deontic normative notions. Inconsistency management in requirements analysis has also been investigated [11, 12]; this work examines how useful the description of all system behaviour by means of a causal vocabulary would be in detecting conflict. However, this work is primarily based on causal logic which does not allow us to easily define and reason about the deontic and temporal aspects of policy. This is significant since the specification of the deontic and temporal characteristics of policy has been integral in detecting dynamic, potential conflict.

This paper begins by providing an overview of our policy model, focusing on the definition and specification of policy in Section 2. A classification of conflicts and discussion of the static and dynamic nature of such conflicts follows in Section 3 and Section 4 describes our method for managing conflict in a computationally-efficient manner. We conclude the paper with a discussion of further work in Section 5.

## 2 Policy Model

Due to the continually evolving nature of enterprises, it is imperative that systems be able to adapt to and accommodate increasingly complex requirements without the need for substantial changes to system structure or application algorithms. Furthermore, enterprise systems need to be able to cope with the extraordinarily high rate of change ubiquitous in such environments, for example, merging distributed computing domains which often results in the introduction of new services and a need for heterogenous applications to co-operate.

In our previous work, we have developed a model of policy-based management for supporting the requirements of large evolving enterprises [3]. This section begins by providing an overview of the concepts within our policy model, as introduced in [3], followed by a detailed discussion of our definition of policy, which is pertinent to addressing the issue of dynamic conflict detection and resolution.

### 2.1 Overview of Policy Model Concepts

Central to our model are the concepts of *Enterprise Domain*, *Policy Space*, *Role* and *Policy Authority*, represented in the simplified model of Figure 1. Enterprise domains are enterprise-level classifications of all the entities (users, roles, artefacts) within the organisation and are a convenient means by which we can *enumerate* objects under common authority or ownership. As such, it is typically, but

not always, a hierarchical division of departments, faculties, teams etc.

The purpose of identifying policy spaces, on the other hand, is to combine all of the entities and actions about which we *are interested in writing and applying policy*. A policy space is generally a declarative statement or description of the entities and actions about which policy will be written. Policy spaces are defined as distinct from enterprise domains because the application of policy typically does not follow clean, organisational boundaries. That is, where an enterprise domain might be *Computer Science Department*, a policy space might be *University Parents*, where *University Parents* would be a selection of entities belonging to many disparate enterprise domains within the organisation, possibly including a selection of those from the *Computer Science Department*.

The central premise of policy-based management is that users do not have indiscriminate access to enterprise objects; instead permission to access enterprise objects and duty to perform assigned organisational obligations (*policy*) are associated with roles. Entities can then be assigned to roles as determined by their responsibilities and qualifications. Roles in our model may be either static or dynamic. The difference being, that membership to static roles is an enumeration of entities within the organisation, whereas membership to dynamic roles is evaluated at run-time according to some predefined predicate (for example, *Cheapest Supplier*).

Policy authorities in our model are assigned to both enterprise domains and policy spaces. An enterprise domain authority is the assigned owner of resources within the enterprise domain, whereas a policy space authority is one which may write policy regarding the entities and actions contained within the policy space. Recognising policy authorities is essential for conflict resolution.

### 2.2 Definition of Policy

In this section we examine the deontic concepts of *permission*, *prohibition* and *obligation* broadly used to define policies. We give an overview of the notation we use to describe policies and provide an example of a formal temporal specification of the policy types; which together have been pivotal in allowing us to reason about states of conflict and to consequently define appropriate methods for dynamic conflict detection and resolution.

We believe the purpose of *policy* is to define or constrain the current or future behaviour of objects to ensure that their actions are aligned with the objectives of the enterprise. Initially, policies are specified at an enterprise-level and represent statements about the organisations requirements and goals. These statements are usually specified in some abstract natural language (*for example*, structured english) and

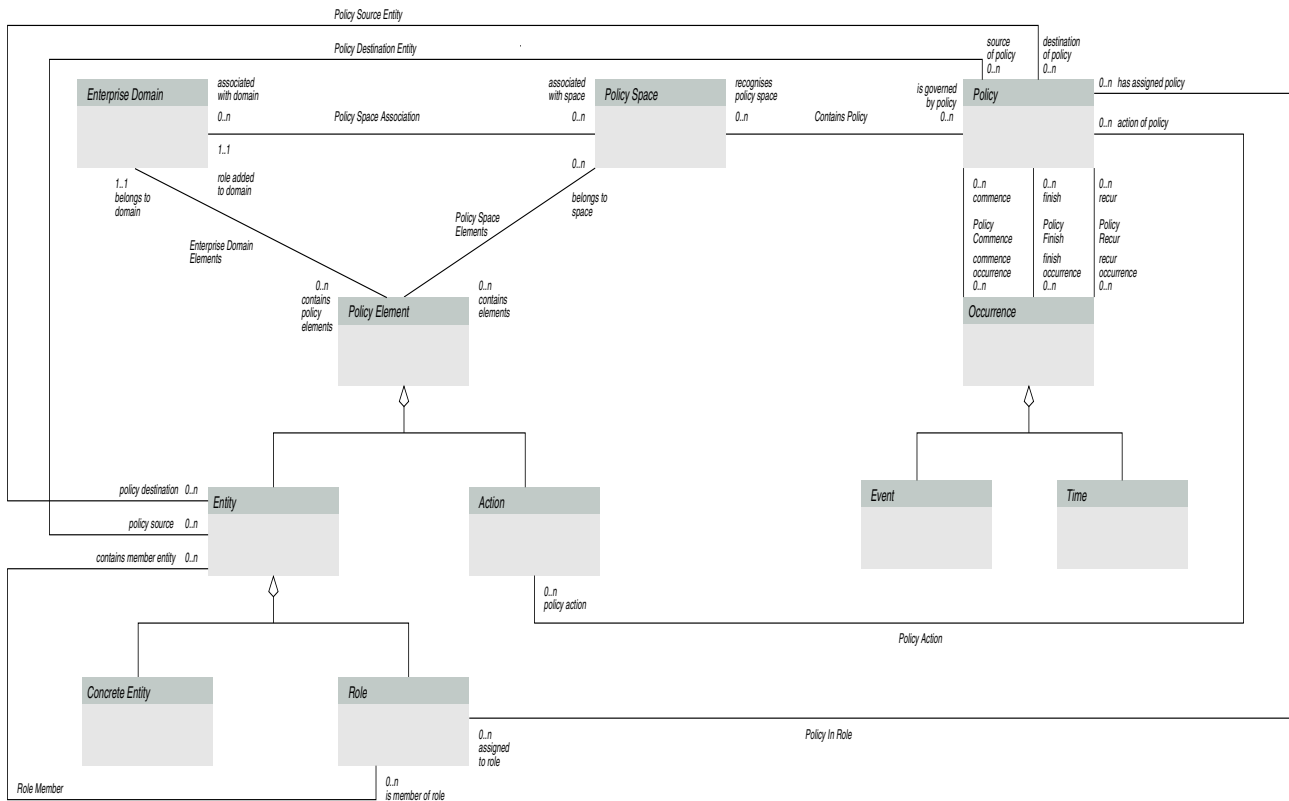


Figure 1. Partial Dynamic Policy Model Represented in the Unified Modeling Language (UML).

are then refined into more concrete (*or executable*) statements about organisational behaviour or *policy*.

### 2.2.1 Deontic Nature of Policy

While deontic logic expresses many normative notions that could be useful in policy specification, for example, *rights, claims, liberties, privileges, power, immunity, prima facie, conditional and defeasible obligations, contrary to duty* etc; we choose to focus on the set of modal operators denoting states of *obligation, permission* and *prohibition*, as defined in standard deontic logic [6] and widely accepted in the literature [2, 5, 7, 15]. These policy modes can be described as follows:

- *Permission* is “the action of permitting or giving leave; allowance; liberty or a licence granted to do something” [16].
- *Prohibition* is “an unambiguous statement or rule, regulating *forbidden* behaviour in a system” [14].
- *Obligation* is “an agreement, enforceable by law, whereby a person or persons become bound to a particular action or performance of some duty by a contract containing such an agreement” [14].

### 2.2.2 Policy Notation

The purpose of this section is to provide a class description of policy, Figure 2 and discussion of its associated terminology, which together will form the foundation of our later discussion of dynamic conflict detection.

For the most part, the meaning of the attributes in the above policy class definition are readily apparent and will be briefly discussed below. What is not necessarily apparent, however, are the references to the temporal characteristics of policy. We believe that an understanding of the temporal nature of policy is pivotal to both the specification and subsequent classification of conflict in policy-based management systems.

To this end, we have defined a number of novel policy types (based on the deontic concepts of *obligation, permission* and *prohibition*). These policy types are formally specified in Section 2.3, but for now we will state that:

- *PolicyName* is used to uniquely identify each policy.
- *PolicyDescription* is a textual interpretation of the purpose of the policy.
- *PolicyType* refers to the deontic notions of *obligation (O+, O-), permission (P)* and *prohibition (F)*.

```

class Policy {
  attribute string PolicyName;
  attribute string PolicyDescription;

  attribute char PolicyType;
  reference Subject to policy_subject
    of PolicySubjectEntity;
  reference Action to policy_action of PolicyAction;
  reference Target to policy_target
    of PolicyTargetEntity;

  attribute string TemporalClassifier;
  reference Commence to commence_occurrence
    of PolicyCommence;
  reference Finish to finish_occurrence of PolicyFinish;
  reference Recur to recur_occurrence of PolicyRecur; };

```

**Figure 2. Class Description of Policy.**

- *TemporalClassifier* denotes one of the policy types belonging to either *obligation*, *permission* or *prohibition*.

That is, a temporal classifier might indicate that an obligation holds always, at all times, as opposed to an obligation that recurs at a stated time or event. For example,  $[O_{always}]$  defined in Section 2.3, denotes a continuous obligation that holds always, at all times.

- *Commence* is used to identify an instance of the type Occurrence (event or time), at which this policy will commence.
- *Finish* is used to identify an instance of the type Occurrence (event or time), at which this policy will be dismissed.
- *Recur* is used to identify an instance of the type Occurrence (event or time), at which this policy will recur.

## 2.3 Formal Specification of Policy

We have chosen to formally specify *obligation*, *permission* and *prohibition* policies in a temporal logic language that allows us to reason about sequences of events [1, 9, 10]. In order to illustrate how we have formally defined our policy types, we include a description of the notation used within the specification in Section 2.3.1, followed by the semantics of the temporal operators employed in Section 2.3.2 and conclude with an example of the obligation policy specification in Section 2.3.3.

### 2.3.1 Specification Notation

In this example specification, the following notation is used -  $\phi, \psi, \delta \in \text{event/time}$  and the sequence  $\sigma$  which represents a history of these *event/time* points; and the notation  $\alpha \in \text{action}$ .

We also introduce the operators  $\text{DO}(\alpha)$  - *action occurs next* and  $\text{DONE}(\alpha)$  - *action occurred previously*. Both operators occur relative to some nominated point in the history  $\sigma$ , over the set of actions  $\alpha$ . The semantics of the  $\text{DO}(\alpha)$  operator is defined by:

$$(\sigma, j) \models \text{DO}(\alpha) \text{ iff } (\sigma, j+1) \models \alpha \quad (1)$$

Thus,  $\text{DO}(\alpha)$  holds at position  $j$  iff  $\alpha$  occurs at the next position ( $j+1$ ). While the semantics of the  $\text{DONE}(\alpha)$  operator is defined by:

$$(\sigma, j) \models \text{DONE}(\alpha) \text{ iff } (j > 0) \wedge (\sigma, j-1) \models \alpha \quad (2)$$

Thus,  $\text{DONE}(\alpha)$  holds at position  $j$  iff  $j$  is not the first position in the sequence  $\sigma$  and  $\alpha$  occurred at position ( $j-1$ ). In particular,  $\text{DONE}(\alpha)$  is false at position 0. Therefore, if  $\text{DO}(\alpha)$  holds at position  $j$ , then any point in the history  $\sigma$  after position  $j$ , indicates that  $\text{DONE}(\alpha)$  will be true.

### 2.3.2 Semantics of the Temporal Operators

The semantics of the temporal formula,  $\Box p$ , read *henceforth p* or *always p*, is defined by (3). Thus,  $\Box p$  holds at position  $j$  iff  $p$  holds at position  $k$ , and *all* following positions - “from now on”, in sequence  $\sigma$ .

$$(\sigma, j) \models \Box p \text{ iff } (\sigma, k) \models p \text{ for all } k \geq j. \quad (3)$$

The semantics of the temporal formula,  $\bigcirc p$ , read *next p*, is defined in (4). Thus,  $\bigcirc p$  holds at position  $j$  iff  $p$  holds at the next position  $j+1$ .

$$(\sigma, j) \models \bigcirc p \text{ iff } (\sigma, j+1) \models p \quad (4)$$

The *until* formula,  $p \mu q$ , read *p until q* predicts the eventual occurrence of  $q$  and states that  $p$  holds continuously at least until the (first) occurrence of  $q$ . The semantics of *p until q*, is defined in (5).

$$\begin{aligned}
& (\sigma, j) \models p \mu q \text{ iff} \\
& \text{there exists some } k \geq j, \text{ such that } (\sigma, k) \models q, \\
& \text{and for every } i, j \leq i < k, (\sigma, i) \models p. \quad (5)
\end{aligned}$$

The semantics of the temporal formula,  $\ominus p$ , read *previously p*, is defined in (6). Thus,  $\ominus p$  holds at position  $j$  iff

$j$  is not the first position in the sequence  $\sigma$  and  $p$  holds at position  $(j-1)$ . In particular,  $\ominus p$  is false at position 0.

$$(\sigma, j) \models \ominus p \text{ iff } (j > 0) \text{ and } (\sigma, j-1) \models p. \quad (6)$$

The semantics of the temporal formula,  $\diamond p$ , read *eventually*  $p$ , is defined by (7). Thus,  $\diamond p$  holds at position  $j$  iff  $p$  holds at some position  $k \geq j$ .

$$(\sigma, j) \models \diamond p \text{ iff } (\sigma, k) \models p \text{ for some } k \geq j. \quad (7)$$

### 2.3.3 Example - Obligation Policy Specification

To illustrate how we have defined the policy types, we include an example of an *obligation* policy specification in Figure 3.

Obligation Policy	
<i>Continuous</i>	
$[O_{always}] \cdot always$ - obligation holds at all times.	
$O_{always}(\alpha) \equiv O(\Box(DO(\alpha)))$	(8)
<i>Discrete (true only once)</i>	
$[O_{immediate}] \cdot immediate$ - obligation be performed immediately at state (time or event).	
$O_{immediate}(\phi < \alpha) \equiv [\phi \rightarrow O(\Box(DO(\alpha)))] \mu \ominus DONE(\alpha)$	(9)
$[O_{sometime}] \cdot sometime$ in the future - obligation must be performed at some time in the future.	
$O_{sometime}(\alpha) \equiv O(\diamond(DO(\alpha))) \mu \ominus DONE(\alpha)$	(10)
$[O_{after}] \cdot from$ some future point onwards - obligation must be performed from some future (time or event).	
$O_{after}(\phi < \alpha) \equiv [\phi \rightarrow O(\diamond(DO(\alpha)))] \mu \ominus DONE(\alpha)$	(11)
$[O_{before}] \cdot before$ deadline - obligation must be performed before the stated deadline of (time or event).	
$O_{before}(\alpha < \psi) \equiv O(\neg\psi \mu DO(\alpha)) \mu (\ominus\psi \vee \ominus DONE(\alpha))$	(12)

$[O_{within}] \cdot within$  a period - obligation must be performed within the stated period (x, y) of (time or event).

$$O_{within}(\phi < \alpha < \psi) \equiv ([\phi \rightarrow O(\neg\psi \mu DO(\alpha))]) \mu ((\ominus\phi) \vee (\ominus DONE(\alpha))) \quad (13)$$

*Recurrent (periodically occur)*

$[O_{recur\_always}] \cdot recur$  always - obligation recurs periodically at (time or event) for all time.

$$O_{recur\_always}(\phi < \alpha) \equiv \Box[\phi \rightarrow O(\Box \Box (DO(\alpha)))] \quad (14)$$

$[O_{recur\_after}] \cdot recur$  from some future point onwards - obligation recurs periodically at (time or event) from some future point onwards.

$$O_{recur\_after}(\phi < \psi < \alpha) \equiv \phi \rightarrow \Box[\psi \rightarrow O(\Box \Box (DO(\alpha)))] \quad (15)$$

$[O_{recur\_until}] \cdot recur$  until deadline - obligation recurs periodically at stated (time or event) until deadline.

$$O_{recur\_until}(\phi < \alpha < \psi) \equiv (\Box[\phi \rightarrow O(\Box \Box (DO(\alpha)))] \mu \ominus \psi) \quad (16)$$

$[O_{recur\_within}] \cdot recur$  within a period - obligation recurs periodically at (time or event) within the stated period (x, y).

$$O_{recur\_within}(\phi < \alpha < \psi) \equiv (\phi \rightarrow (\Box[\delta \rightarrow O(\Box \Box (DO(\alpha)))])) \mu \ominus \psi \quad (17)$$

**Figure 3. Formal Specification of Obligation Policy Types**

## 3 Conflict Detection

Policy conflict occurs when the objectives of two or more policies cannot be simultaneously met. However, detecting and resolving this conflict is a significant research challenge. Current advances in conflict detection and resolution have been primarily focused on static, compile-time environments [8]. There exists, however, a compelling requirement to examine the very complex and unresolved issues surrounding dynamic (or run-time) conflict detection and resolution. In this Section we describe a classification of conflicts and discuss the need for both dynamic and static conflict analysis.

### 3.1 Classification of Conflicts

We believe conflicts can be classified into four broad categories as defined below. Note that each category of conflict may present itself either statically or dynamically.

#### 3.1.1 Internal Policy Conflict

Internal Policy Conflict occurs when the policies assigned to a *single* role are deemed to be incompatible with each other. Detection of internal policy conflict is required when a new policy is added to the role specification either when the role is initially defined or sometime during the lifetime of the system as the objectives of the role evolve.

#### 3.1.2 External Policy Conflict

External Policy Conflict occurs when a user combines roles, which in isolation of each other present no conflict, but contain policies which in co-existence are in conflict. External policy conflict may be detected when a new user is assigned to a role (this may occur at run-time) and/or when a new policy is assigned to a role.

#### 3.1.3 Policy Space Conflict

Policy Space Conflict occurs when two or more policy spaces manage the same set of subjects and attempt to enforce different and conflicting policies over them. Detection of policy space conflict needs to occur both when the policy spaces are initially identified and at run-time, when a new policy is assigned to a role (which may be in conflict with another policy assigned by a different policy space).

#### 3.1.4 Role Conflict

Role conflict occurs when a user obtains a set of incompatible role assignments. Detection and subsequent resolution of role conflict is required to ensure that the user does not operate with a union of privileges determined to be incompatible. For example, obtaining the roles of *banker* and *auditor* would likely be considered a role conflict. Role conflicts need to be detected both when users are initially assigned to roles and when users acquire roles at run-time.

### 3.2 Static and Dynamic Conflict Detection

It is important to make the distinction between the two broad classes of static and dynamic conflict as conflict detection and resolution can be computationally intensive, time-consuming and hence, expensive and would *preferably* be done statically, at compile-time. Identified *static conflict* therefore requires immediate attention and resolution, as it will most certainly result in conflict at some time. Whereas,

*dynamic, potential conflict* is quite unpredictable, in that it may, or may not, proceed to actual conflict; that is, the inconsistency may be exposed temporarily, or indeed not at all. The goal of conflict detection is therefore:

- to *identify* actual conflict that has occurred and can be resolved statically, at compile-time.
- to *predict* that a conflict, may, occur in the future (and more specifically, exactly what circumstances will expose that conflict)
- to *monitor* identified potential conflicts; and
- to *communicate* the actual or potential conflict to a resolution process or in some cases, a human operator, for assistance in the *resolving* the conflict situation.

Note, however, that not all predicted conflicts require notification or action. For example, a conflict may be predicted to occur, but be far enough into the future or uncertain enough that an alert would be a nuisance and action would not be appropriate at the current time.

Being able to detect conflict statically and at run-time relies on a knowledge of the temporal characteristics of the policies in the specification. Through examining each combination of the policy types we defined in Section 2.3, we have been able to determine when each of the policy types would cause a conflict (actual and/or potential). Note that the complete specification of conflicts occurring between these policy types has been recorded in a conflict database to be described further in Section 3.2.1. Interestingly, the majority of conflicts could not be determined *statically* at specification time, but rather, referred to potential conflicts that could only be determined *dynamically* at run-time.

For example, if we were to define a role containing two policies *p1* and *p2*, where *p1* has the temporal classifier [*F<sub>recur\_always</sub>*] - *prohibition recurring at stated time or event, for all time*, defined as:

$$F_{recur\_always}(\phi < \alpha) \equiv \Box[\phi \rightarrow (\Box \circ O(\neg\alpha))] \quad (18)$$

while policy *p2* has the temporal classifier [*O<sub>within</sub>*] - *obligation must be performed within a stated period ( $\phi, \psi$ ) of time or event*, and both *p1* and *p2* have an overlapping scope specified by the following pre-condition:

$$\begin{aligned} & ((P_x.Subject \cap P_y.Subject \neq \emptyset) \wedge \\ & (P_x.Action \cap P_y.Action \neq \emptyset) \wedge \\ & (P_x.Target \cap P_y.Target \neq \emptyset)) \end{aligned} \quad (19)$$

we would discover (through examination of the conflict database) that these policies exhibited actual and potential *internal policy conflicts* as described in the conflict profile

below. It should be noted that, throughout Figure 4, the notation (*C*) refers to the *Commence* attribute of the policy class definition in Figure 2 and similarly, (*F*) refers to the *Finish* attribute and (*R*) refers to the *Recur* attribute.

---



---

$p1:[F_{recur\_always}] \cdot recur \text{ always}$ $p2:[O_{within}] \cdot within \text{ a period}$	
--	--

---

**Actual Conflict**

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{time} \text{ between } [p2.C_{time}, p2.F_{time}] \quad (20)$$

$$(\forall p2.C_{time}, p2.F_{time}, p1.R_{time} : time |$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{time} \text{ between } [p2.C_{time}, p2.F_{time}]) \quad (21)$$

---

**Potential Conflict**

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{time} \text{ between } [p2.C_{event}, p2.F_{event}] \quad (22)$$

$$(\forall p2.F_{time}, p1.R_{time} : time, p2.C_{event} : event |$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{time} \text{ between } [p2.C_{event}, p2.F_{event}]$$

$$\text{ where trigger } [p2.C_{event}]) \quad (23)$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{event} \text{ between } [p2.C_{time}, p2.F_{time}] \quad (24)$$

$$(\forall p2.C_{time}, p2.F_{time} : time, p1.R_{event} : event |$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{event} \text{ between } [p2.C_{time}, p2.F_{time}])$$

$$\text{ where trigger } [p2.C_{time}] \quad (25)$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{event} \text{ between } [p2.C_{event}, p2.F_{time}] \quad (26)$$

$$(\forall p2.F_{time} : time, p2.C_{event}, p1.R_{event} : event |$$

$$p1 \xrightarrow{\text{conflicts}} p2 \text{ at } \forall p1.R_{event} \text{ between } [p2.C_{event}, p2.F_{time}])$$

$$\text{ where trigger } [p2.C_{event}] \quad (27)$$

---



---

**Figure 4. Conflict Database Entry.**

Accordingly, from (20), we understand that if policy *p1* has a *recur* attribute that is of the occurrence type, time, and policy *p2* has both *commence* and *finish* attributes of the occurrence type, time; then we will need to further assess for **actual** conflict (coinciding with every recurrence of *p1.Rtime* within the period referenced by [*p2.Ctime*, *p2.Ftime*]).

If, however, policy *p1* has a *recur* attribute that is of the occurrence type, event, and policy *p2* has both *commence* and *finish* attributes of the occurrence type, event; then commensurate with (27), we will need to manage the **potential** for conflict (occurring at every recurrence of *p1.Revent* within the period referenced by [*p2.Cevent*, *p2.Fevent*]). It is clear that policies of these particular occurrence types will need to be managed at run-time, as the realisation of conflict is dependent on the incidence and pattern of events occurring at run-time.

Thus, the types of the temporal attributes, *commence*, *finish* and *recur*, inevitably dictate the type of conflict that will need to be managed (ie. actual or potential).

### 3.2.1 Conflict Database

Through the course of our work, we have defined a conflict database [4], representing all possible conflicts that may occur for the combination of policy types described in Section 2.3. The results of producing this database revealed an overwhelming majority of dynamic conflicts requiring management, that is, of the 4,238 types of conflict identified, 3,850 (or 90.84%) were dynamic conflicts only able to be managed at run-time, and a mere 388 were static conflicts or (9.16%) able to be managed at compile-time. Using the profiles in the conflict database it is possible to not only detect actual, static conflict, but also to understand the nature of potential, dynamic conflicts and under what specific run-time circumstances they will present themselves. It should be noted that aspects of this conflict database are able to be generated automatically.

### 3.2.2 Conflict Example

As an example, if we were to introduce the following policies [*p1*, *p2*, *p3*, *p4*, *p5*], then using the conflict database, we would be able to predict that the policies of *p1* and *p3* would be in a state of potential conflict between [*Time2*, *Time3*], see Figure 5. This is because policy *p1* becomes active when (*Revenue* < \$15,000), and will remain active until (*Revenue* ≥ \$15,000); while policy *p3* becomes active at every occurrence of the event (*Profit* < \$5,000). Therefore, when the event (*Revenue* < \$15,000) occurs, we need to be observant of the event (*Profit* < \$5,000), as the occurrence of this particular event will mean that the conflict between *p1* and *p3* has been realised. In this example, conflict is realised at Time 3 when this event occurs, see Figure 5.

Note, however, that dependent on the pattern of run-time events, the policies of  $p1$  and  $p3$  may *never* proceed to realised conflict. An example of a pattern of run-time events that will not lead to conflict is detailed in Figure 6. Here we know that policy  $p1$  is active when ( $Revenue < \$15,000$ ), which commences at Time 2, until ( $Revenue \geq \$15,000$ ). We also know that if policy  $p3$  becomes active after Time 2 and before ( $Revenue \geq \$15,000$ ), at its recurrence event of ( $Profit < \$5,000$ ), then conflict will be realised. We are therefore, watchful of the event ( $Profit < \$5,000$ ), however, in this case,  $p1$ 's active state closes at the event ( $Revenue \geq \$15,000$ ) which occurs at Time 3, that is, before the event ( $Profit < \$5,000$ ) has had a chance to occur. These policies have not resulted in run-time conflict requiring resolution.

The central question then becomes how to manage these potential, run-time conflicts in a way that is computationally-efficient and is the subject of Section 4.

```
Policy "p1" {
PolicyName: "Chief Purchasing Officer - Revise Strategy Duty"
PolicyDescription: CPO - required to revise strategy when
revenue level is unacceptable (<$15,000).
PolicyType: "O+"
Subject: StaticRole "Chief Purchasing Officer"
Action: Action "Revise Plan"
Target: Artefact "Organisational Strategy"
TemporalClassifier: "Owithin"
Commence: ExternalEvent "Revenue < $15,000"
Finish: ExternalEvent "Revenue ≥ $15,000" }
```

```
Policy "p2" {
PolicyName: "Chief Purchasing Officer - Revise Strategy
Authorisation"
PolicyDescription: CPO - authorised to revise strategy when
revenue level is unacceptable.
PolicyType: "P"
Subject: StaticRole "Chief Purchasing Officer"
Action: Action "Revise Plan"
Target: Artefact "Organisational Strategy"
TemporalClassifier: "Pwithin"
Commence: ExternalEvent "Revenue < $15,000"
Finish: ExternalEvent "Revenue ≥ $15,000" }
```

```
Policy "p3" {
PolicyName: "Chief Purchasing Officer - Revise Strategy
Prohibition"
PolicyDescription: CPO - cannot revise strategy when
profit level is unacceptable (<$5,000).
PolicyType: "F"
Subject: StaticRole "Chief Purchasing Officer"
Action: Action "Revise Plan"
Target: Artefact "Organisational Strategy"
TemporalClassifier: "Frecur_always" }
```

```
Recur: ExternalEvent "Profit < $5,000" }
```

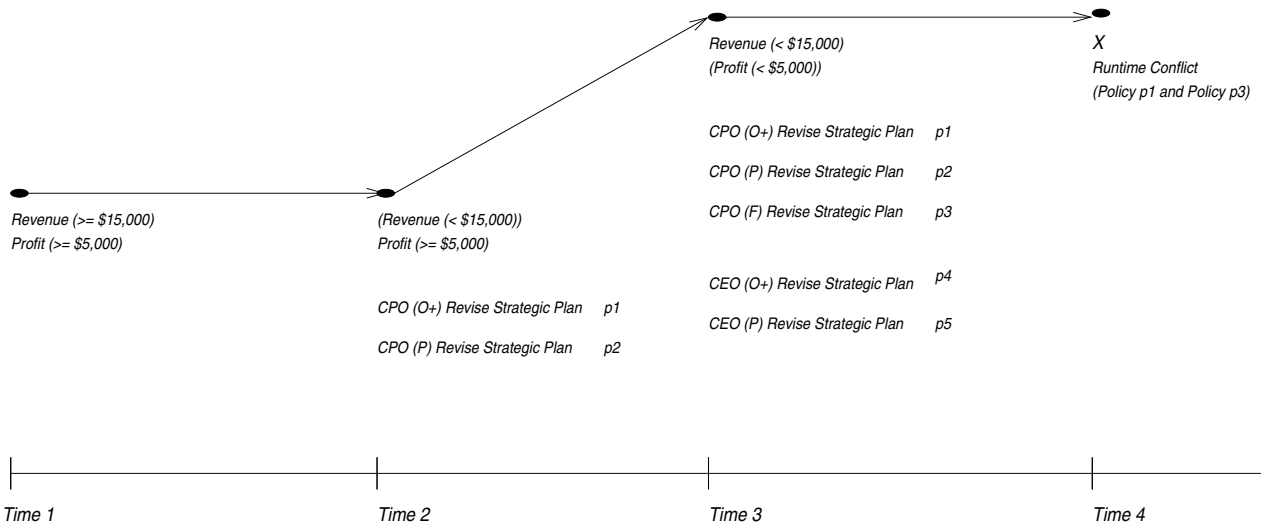
```
Policy "p4" {
PolicyName: "Chief Executive Officer - Revise Strategy Duty"
PolicyDescription: CEO - required to revise strategy when
profit level is unacceptable (<$5,000).
PolicyType: "O+"
Subject: StaticRole "Chief Executive Officer"
Action: Action "Revise Plan"
Target: Artefact "Organisational Strategy"
TemporalClassifier: "Owithin"
Commence: ExternalEvent "Profit < $5,000"
Finish: ExternalEvent "Profit ≥ $5,000" }
```

```
Policy "p5" {
PolicyName: "Chief Executive Officer - Revise Strategy
Authorisation"
PolicyDescription: CEO - permitted to revise strategy when
profit level is unacceptable.
PolicyType: "P"
Subject: StaticRole "Chief Executive Officer"
Action: Action "Revise Plan"
Target: Artefact "Organisational Strategy"
TemporalClassifier: "Pwithin"
Commence: ExternalEvent "Profit < $5,000"
Finish: ExternalEvent "Profit ≥ $5,000" }
```

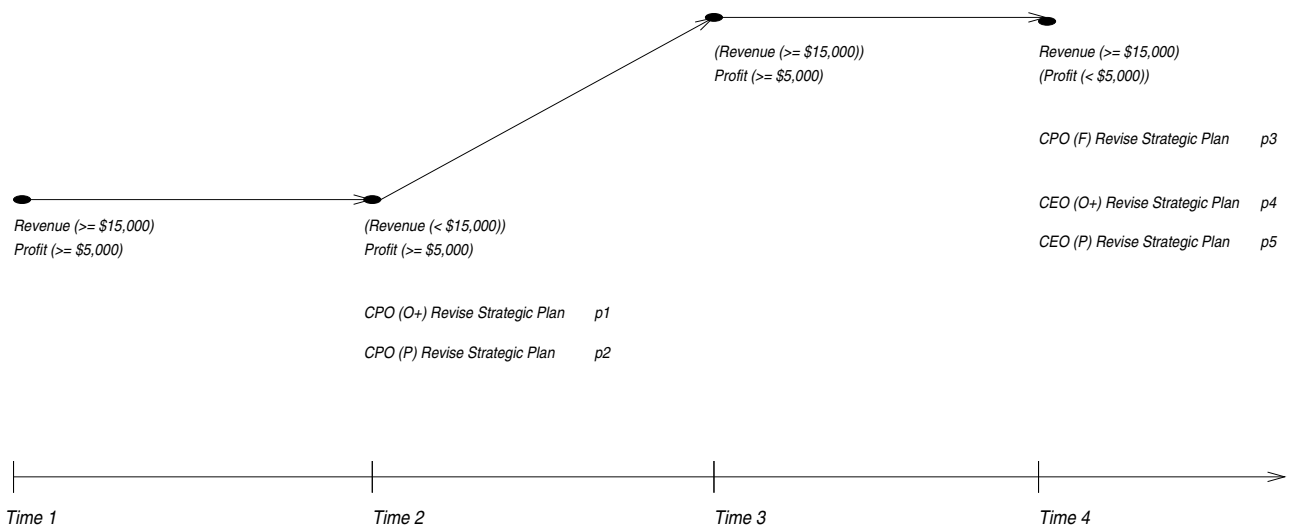
## 4 Scalable Dynamic Conflict Detection

Conflict detection and resolution at run-time occurs as a result of certain run-time events. It should be noted that the type of run-time event occurring dictates what sort of conflict detection and resolution is appropriate. The types of run-time event that necessitate conflict detection and resolution can be characterised as follows:

- *add policy* event occurs when a user attempts to add a new policy to an existing pre-defined role.
- *action attempt* event occurs when a user attempts some defined action (for example, attempting to order stock) that requires us to evaluate the index of potential conflicts for changes in status (that is, a user attempting to perform some action may cause a run-time conflict to progress from *inactive* or *pending* to *active*).
- *entity change* event is interesting as modification of certain entity properties, for example, the modification of the predicate governing the calculation of membership in a dynamic role would require the reanalysis of membership to that role. We also know that if role memberships change, we will need to examine the modified roles for newly initiated conflict, usually role conflict and/or external policy conflict.



**Figure 5. Dynamic Conflict Detected at Time 3.**



**Figure 6. No Dynamic Conflict Detected.**

- *assign entity to role* event requires an analysis of the potential emergence of role conflict and external policy conflict, which again, may occur statically and/or dynamically.
- *external event* require us to check a run-time index of potential conflicts for changes in status (that is, external events frequently result in run-time conflict progressing from *inactive* or *pending* to *active* conflict).

Following in Section 4.1 we describe the algorithms used to manage conflict. Section 4.2 will illustrate the use of the algorithms, as applied to the example of the event *add policy*.

#### 4.1 Algorithms for Conflict Detection

Because each of the run-time occurrences listed above require quite specific and involved conflict detection and resolution routines, we will describe a subset of algorithms we use for run-time conflict detection. For the purposes of this paper, we include the specific routines of *dynamic\_conflict\_management*, *check\_internal\_policy\_conflict* and *process\_external\_event*, followed by their associated routines *record\_run-time\_conflict* and *examine\_run-time\_conflict\_index*. The algorithms involved in examining external policy conflict, policy space conflict and role conflict are not considered within the scope of this paper.

This section commences with a discussion of the indexed databases we use to assist in dynamic conflict detection. Followed by a description of the dynamic conflict detection algorithms.

#### 4.1.1 Indexed Event Databases

Central to the development of the scalable conflict detection algorithms are two indexed databases; the *active events index* and the *run-time conflict index*. The *active events index* contains the tuples  $\langle event, status \rangle$  which represent a list of run-time events and their current status as either *active* or *inactive*. The *run-time conflict index*, contains the tuples  $\langle policy\_references, initiate\_event, activate\_event, dismiss\_event, status \rangle$  which reflects an indexed list of current potential run-time conflicts. The attribute *policy\_references* indicates which policies will be in conflict, *initiate\_event* refers to the event that will force a conflict from the status of “inactive” to “pending”. The *activate\_event* forces a conflict from a state of “pending” to a state of “active” conflict and *dismiss\_event* dissolves the conflict state to “inactive”. Status is either *active*, *pending* or *inactive*.

#### 4.1.2 Description of Conflict Detection Algorithms

In the situation where run-time events lead us to a need to examine policies for conflict, we will initially check the policies for adherence to the intersecting concerns precondition specified in (19), Section 3.2. We then extract the temporal classifiers of the policies in question and look up the conflict database to see what sort of conflict profile policies of these particular temporal characteristics reveal. Once we have retrieved the conflict profile, we include the potential conflicts between these policies in the run-time conflict index, describing precisely which events will initiate, activate and dismiss this conflict.

The advantage of detailing potential conflicts in an indexed list of this nature and in specifying the conflict database, is that when events are occurring at run-time, we need only look up the index to see which conflicts will be affected by the occurrence of such an event, and do not need to examine every combination of active policies and events at run-time to determine if the occurring event produces a conflict.

*begin – dynamic conflict management()*

```
switch (user_activity) {
  case 'add policy' : {
    call check_internal_policy_conflict(policy, role);
    call check_external_policy_conflict(policy, role);
    call check_policy_space_conflict(policy, role); }
  case 'action attempt' : {
    call examine_runtime_conflict_index(policy, role); }
```

```
case 'entity change' : {
  call check_role_conflict(policy, role); }
call check_external_policy_conflict(policy, role); }
case 'assign entity to role' : {
  call check_role_conflict(policy, role); }
call check_external_policy_conflict(policy, role); }
case 'external event' : {
  call process_external_event(event); }
```

} end switch – user activity

end – dynamic conflict management

*begin – check internal policy conflict(policy new\_policy, role r)*

with (subject, action, target, policy\_type) of new\_policy;

```
for (each existing_policy in role (r)) {
  with (subject, action, target, policy_type) of existing_policy;
  determine overlap (subject, action, target) of
  new_policy and existing_policy;
```

if there exists overlap {

```
  if policy_types of new_policy and existing_policy
  reflect (O+/O-), (P/F) or (O+/F) {
    with (temporal_classifier) of
    new_policy and existing_policy;
```

```
  with conflict profile of the policy_types
  (new_policy.temporal_classifier,
  existing_policy.temporal_classifier)
  from conflict database;
```

```
  if conflict profile reveals actual conflict {
    report the actual internal policy conflicts to
    a conflict resolution process;
  } end if – actual conflict
```

```
  else if conflict profile reveals potential conflict{
    call record_run-time_conflict(new_policy,
    existing_policy, initiate_event,
    activate_event, dismiss_event);
  } end else if – potential conflict
```

} end if – conflicting policy types

} end if – there exists overlap

} end for – existing policies

end – check internal policy conflict

*begin – record run-time conflict(policy p1, policy p2, event initiate, event activate, event dismiss)*

```
insert tuple <p1, p2, initiate, activate, dismiss, “Inactive”>
into run-time_conflict_index;
```

end – record run-time conflict

*begin – process external event(event e)*

```
update tuple <e, status> to <e, “Active”>
```

```

    in active_events_index;
    call examine_run-time_conflict_index(e);

```

*end – process external event*

*begin – examine run-time conflict index(event e)*

```

for (each tuple in index with initiate field = (e)) {
    update conflict status to "pending"
    in runtime_conflict_index;
} end for – each initiate event in index

for (each tuple in index with activate field = (e)) {
    update conflict status to "active"
    in runtime_conflict_index;
    report active conflict to a conflict resolution process;
} end for – each activate event in index

for (each tuple in index with dismiss field = (e)) {
    update conflict status to "inactive"
    in runtime_conflict_index;
} end for – each dismiss event in index

```

*end – examine run-time conflict index*

## 4.2 Method for Scalable Dynamic Conflict Detection Using Example of Add Policy

In order to describe how the conflict detection algorithms are used, we will focus on the occurrence of event *add policy*. We also use the policies introduced in Section 3, in particular, policies *p1* and *p3*, in order to describe how dynamic conflict detection is efficiently managed. For the purposes of this example, let us assume that policy *p1* is currently assigned to the role of “Chief Purchasing Officer”. Let us assume that an authority then decides to introduce policy *p3* to the role “Chief Purchasing Officer” at run-time. The definition of this *Add Policy* event is as follows:

```

AddPolicy “Chief Purchasing Officer - Strategy Prohibition”
{
AddPolicyReference: “Chief Purchasing Officer - Strategy Prohibition”
PolicyToAdd: Policy: “p3”
PolicyRoleAssignment: StaticRole “Chief Purchasing Officer”
PolicySpaceAssignment: PolicySpace “Purchasing”
}

```

Before this new policy can be added to the system, the conflict detection routines will need to assess whether adding this new policy will result in actual and/or potential conflict of the following:

- *internal policy conflict* it can be seen from the *add policy* definition above, that assessing for internal policy conflict will require examining whether adding *p3* to

the role definition of “Chief Purchasing Officer” will produce a conflict state.

- *external policy conflict* examining external policy conflict requires us to derive the set of all roles held by the “Chief Purchasing Officer” in order to determine if the new policy *p3* causes conflict with any of the other policies assigned to currently held roles.
- *policy space conflict* involves determining if there exists another policy space aside of “Purchasing” that manages the same set of subjects referenced in the Subject, Action and Target attributes of *p3*. If there does exist another policy space, then we will need to further determine if this policy space has issued any policy over the set of Subject, Action and Target and indeed if these policies are inconsistent.

It is important to note that each of these types of conflict may present themselves as either static and/or dynamic conflict. For example, we know that the system needs to assess whether adding the policy *p3* to the role of “Chief Purchasing Officer” will produce an internal policy conflict. The task is to obtain the set of all policies related to the role “Chief Purchasing Officer” and examine each one against the new policy to be added to determine if there is equivalence between their Subject, Action and Target attributes. In this example, it would be determined that the policy *p1* is currently assigned to the “Chief Purchasing Officer” and it does indeed refer to the same Subject, Action and Target attributes as policy *p3*. Consequently, determination as to whether these types of policies (that is,  $F_{recur\_always}$  and  $O_{within}$ ) would cause actual or potential conflict with one another is then required.

This would be achieved through looking up the conflict database, which would in fact reveal that the policy types of *p1* and *p3* are associated with the conflict profile (as defined in Section 3.2). We further determine from the specified conflict profile, that these two policies exhibit potential, dynamic conflict in the following circumstance:

$$(\forall p2.C_{event}, p2.F_{event}, p1.R_{event} : event \mid p1 \xrightarrow{\text{conflict}} p2 \\
 \text{at } \forall p1.R_{event} \text{ between } [p2.C_{event}, p2.F_{event}] \\
 \text{where trigger } [p2.C_{event}]) \quad (28)$$

Which can be interpreted as,

$$p1 \xrightarrow{\text{conflict}} p2 \text{ at } \forall (\text{Profit} < \$5,000) \\
 \text{from } (\text{Revenue} < \$15,000) \text{ until } (\text{Revenue} \geq \$15,000) \\
 \text{where trigger } (\text{Revenue} < \$15,000) \quad (29)$$

Given that we know there is a potential for conflict at this time, we include this potential conflict in the *run-time conflict index* to monitor at run-time. Maintaining a list of potential conflict situations means that we need only examine

this index when events occur at run-time to see if the potential conflict is exposed. We do not need to examine every single combination of active policies at run-time to see if the run-time event occurring produces a conflict. This therefore, provides us with a scalable, computationally-efficient solution for dynamic conflict detection.

### 4.3 Prototype

Our work has incorporated the development of a prototype to demonstrate how dynamic conflict is detected and presented. The largely Java-based implementation has incorporated the use of the Object Management Group's Meta-Object Facility (MOF) [21]. The Meta-Object Facility both stores the class definitions of components in our policy model, Figure 1, and stores its associated instances. A Human Usable Textual Notation (HUTN) product was used to initially parse and input the policy instances [19].

## 5 Conclusion and Further Work

The specification of the temporal characteristics of policy has been critical in enabling us to reason about the consistency of policy-based specifications, and more specifically about when and under what circumstances policy-based conflict occurs. We have found that there exists a large class of conflict that cannot be determined statically and that through careful specification of policy types, focusing on the temporal properties of policy, we have been able to classify both static and dynamic conflict and demonstrate that it can be efficiently managed (detected) at run-time. It should also be noted that in our current work we detect *n*-way conflict through detection of the pairwise policy conflicts that must occur between them. In future work we will consider the more direct detection of *n*-way conflict, perhaps further improving the efficiency of our algorithms.

Our current work focuses on conflict resolution techniques. Several techniques are necessary in order to deal with the varied range of policy-based conflicts and the suitability of each technique is dependent on the type of conflict it is attempting to resolve. We have found that because conflict occurs not only within the scope of a single policy space, but also across organisational boundaries, that it becomes imperative to understand the nature of the relationships that exist between policy spaces and enterprise domains (*for example*, senior/dependent, co-operating, independent) [3]. Furthermore, we have found that representing authorities within our policy model has been key to resolving conflict between multiple domains. However, conflict resolution is a sizeable and complex topic that is outside the scope of this paper.

## 6 Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science and Resources).

## References

- [1] M. Chechik, D. O. Păun, "Events in Property Patterns", Theoretical and Practical Aspects of SPIN Model Checking, Lecture Notes in Computer Science, Volume 1680, Springer-Verlag, September, 1999.
- [2] F. Chen, R. S. Sandhu, "Constraints for Role-Based Access Control", Proceedings of the First ACM/NIST Role-Based Access Control Workshop (RBAC '95), Maryland, USA, December, 1995.
- [3] N. Dunlop, J. Indulska, K. A. Raymond, "Dynamic Policy Model for Large Evolving Enterprises", Proceedings of the Fifth International Conference on Enterprise Distributed Object Computing (EDOC 2001), Seattle, Washington, USA, September, 2001.
- [4] N. Dunlop, J. Indulska, K. A. Raymond, "A Formal Specification of Conflicts in Dynamic Policy-Based Management Systems", DSTC Technical Report, CRC for Enterprise Distributed Systems, University of Queensland, August, 2001.
- [5] W. Keith Edwards, "Policies and Roles in Collaborative Applications", Proceedings of the ACM Workshop on Computer Supported Cooperative Work (CSCW '96), Boston, Massachusetts, USA, November, 1996.
- [6] R. Hilpinen (editor), "Deontic Logic : Systematic Readings", D. Reidel Publishing Company, Dordrecht, Holland, 1971.
- [7] E. C. Lupu, D. A. Marriott, M. S. Sloman, N. Yialelis, "A Policy-Based Role Framework for Access Control", Proceedings of the First ACM/NIST Workshop on Role-Based Access Control (RBAC '95), Gaithersburg, Maryland, USA, December, 1995.
- [8] E. C. Lupu, M. S. Sloman, "Conflicts in Policy-Based Distributed Systems Management", IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management, 1999.
- [9] L. Lamport, "The Temporal Logic of Actions", Proceedings of the ACM Transactions on Programming Languages and Systems (ACM TOPLAS), Volume 16, Issue 3, ACM Press, May 1994.
- [10] Z. Manna, A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems : Specification", Volume 1, Springer-Verlag, New York, 1992.
- [11] J. D. Moffett, J. G. Hall, A. C. Coombes, J. A. McDermid, "A Model for a Causal Logic for Requirements Engineering", Volume 1, Number 1, Journal of Requirements Engineering, 1996.
- [12] J. D. Moffett, A. J. Vickers, "Behavioural Conflicts in a Causal Specification", Volume 7, Number 3, Automated Software Engineering, 2000.
- [13] Object Management Group (OMG), "Complete MOF 1.3 Specification (MOF 1.3) formal/00-04-03", OMG Headquarters, 492 Old Connecticut Path, Framingham, Massachusetts, USA, April, 2000.
- [14] Oxford University Press, "Oxford English Dictionary", January, 2000.
- [15] R. S. Sandhu, "Role-Based Access Control", Advances in Computing, Volume 46, Academic Press, 1998.
- [16] C. Schwarz, G. Davidson, A. Seaton, V. Tebbit, W. and R. Chambers Limited and Cambridge University Press, "Chambers Cambridge English Dictionary", Edinburgh, UK, 1998.
- [17] M. S. Sloman, "Management Issues for Distributed Services", Proceedings of the Second IEEE International Workshop on Services in Distributed and Network Environments (SDNE '95), Whistler, British Columbia, Canada, June 1995.
- [18] "Special Issue on Feature Interactions in Telecommunications Systems", IEEE Communications Magazine, Volume 31, Number 8, 1993.
- [19] J. Steel, K. A. Raymond, "Generating Human-Usable Textual Notations for Information Models", Proceedings of the Fifth International Conference on Enterprise Distributed Object Computing (EDOC 2001), Seattle, Washington, USA, September, 2001.
- [20] M. Thomas, "Modelling and Analysing User Views of Telecommunications Services", Feature Interactions in Telecommunications Networks, IOS Press, 1997.