Generic policy conflict handling using a priori models

Bernhard Kempter[1] and Vitalian A. Danciu[2]

[1] Siemens Corporate Technology
bernhard.kempter@siemens.com
[2] Munich Network Management Team*, University of Munich
danciu@mnm-team.org

Abstract. The promise of policy-based management is lessened by the risk of conflicts between policies. Even with careful conception of the policies it is difficult if not impossible to avoid conflicts completely. However, it is in principle possible to detect and resolve conflicts either statically or at runtime. Taking advantage of existing managed systems models it is even possible to detect and resolve policy conflicts not addressed until now. In this paper we present a generic approach to automated policy conflict detection based on existing knowledge about a managed system. We describe a methodology to derive conflict definitions from invariants of managed systems models, and show how these can be used to detect and resolve policy conflicts automatically.

1 Introduction

As organisations grow – be they corporations, educational facilities or governmental agencies – the number of decision-makers within increases. Business goals formulated by different decision-makers are divergent or conflicting in some cases. When these goals are projected onto IT management, these conflicts will manifest as management conflicts. In principle they will result in conflicting actions, independently of the management architecture deployed, or the associated programming paradigm. Conflicting actions lead to unpredictable results: mild effects could be the failure of single tasks, while more serious cases could lead to faultily configured or malfunctioning systems. Since often business goals are implemented by scripting for management tools, the resolution of conflicts is a task performed after the detection of a conflict and needs to be executed by personnel with insight into the management system, thus incurring high cost. As management evolves in the direction of self-managed systems, automation of conflict handling becomes indispensable.

An important approach to pursuing management goals is the derivation of policy from these goals. Policies at an operational, technical level can be enforced by means of a policy architecture that guides the execution of management actions on a distributed system. Policy-based managment is the only management paradigm that allows, plausibly, conflict detection and resolution. In this paper we present a solution to conflict handling in policy-based systems that exploits a priori models of managed systems.

* The authors wish to thank the members of the Munich Network Management Team for helpful discussions and valuable comments on previous versions of this paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, the University of the Federal Armed Forces and the Leibniz Supercomputing Center.

A generic approach to automated resolution of conflicts between obligation policies is still missing, since such conflicts cannot be resolved from the information given in the policies alone. They are dependent on the structure and setup of the managed system. Thus, a model of the managed system is necessary for determining whether a set of policies is in conflict or not.

Driven by the ever increasing complexity of today's systems, organisations create models of their systems. The advent of service-orientation entices them to model systems in detail and as a whole, in contrast to modeling isolated components. Frameworks like the Common Information Model (CIM) [4] provide a base for such efforts. The resulting models describe the nominal state of the deployed system by not only representing attributes of system components but also relations between them, e.g. functional or structural dependencies. Hence, a finished model can be seen as a specification of the managed system in case.

In this paper we demonstrate how such a priori (i.e. existing) models can be leveraged to detect and solve policy conflicts (Section 3). This allows us to address policy conflict types that until now have been inaccessible to automated detection and resolution. The core of the approach is a methodology for deriving reusable, formal conflict definitions from model aspects and management action sets. These definitions yield constraints that allow automated detection of policy conflicts (Section 4). The broad applicability of the methodology is demonstrated by means of a static relationship model for functional dependency. It can also be applied to other static models, such as containment models, as well as dynamic models, e.g. state models. We use the results of the methodology in Section 5, where we show how automated policy conflict detection can be performed and discuss strategies for automated conflict resolution. We present related work regarding policy conflict resolution and selected modeling techniques and standards in Section 6.

2 Models of managed systems

In this section we discuss aspects of object oriented models instrumental to automated conflict detection and resolution. We bootstrap the approach to policy conflict handling by assessing a general work process of an administrator who enforces policy by hand. Based on that motivation we discuss characteristics of model hierarchies that are useful to our conflict handling approach.

2.1 Manual conflict handling

Consider the two obviously conflicting policies shown in Fig. 1: one policy specifies that all terminals be shut down after working hours, the other specifies that security patches should be installed at that time. A human administrator is able to solve the conflict by allowing the patches to be installed before shutting down the terminals. He assigns an explicit ordering to the policy set to be enforced; in consequence, both policies are enforced and a desired result is achieved.

```
policy {
  event { shopCloses }
  target { /terminals }
  action { shutdown() }
}
```

```
policy {
  event { shopCloses }
  target { /terminals }
  action { update(secPatch)}
}
```

Fig. 1. Simple conflict

In order to find this solution, the administrator uses his knowledge about the managed system: he knows beforehand that patches cannot be installed after the terminals have been shut down. By means of this *a-priori model of the system* he can conclude that he encountered a policy conflict. He uses this model to find an alternative execution path, thus resolving the conflict.

2.2 Hierarchy of models

To describe systems, we usually model them at some level of abstraction. The models normally cover specific static or dynamic aspects of the system they represent, e.g. states and transitions in that system, its structure or its attributes. Hence, models constitute views from different perspectives onto the system.

A given managed object (MO) is embedded in different kinds of models, e.g. it can be a node of a containment tree and, at the same time, a partition of an automaton describing states of the system.

Fig. 2 gives an example of the abstraction hierarchy of models belonging to or derived from CIM. Traversing the diagram from its top downwards, the level of abstraction decreases and the size of models increases. In common practice, a standard like CIM defines abstract classes and associations that are independent of any managed system or vendors (CIM core schema). The core schema is specialized into customized models (often including some levels of abstraction in the customization as well) to fulfill the requirements of an organisation. The resulting customized model is then instantiated according to the infrastructure it represents.

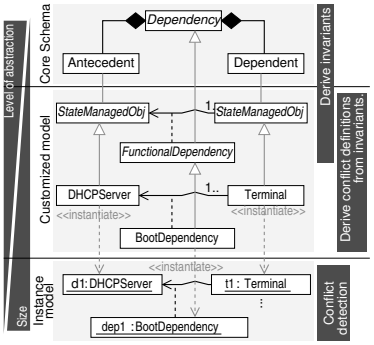

Fig. 2. Hierarchy of models

The diagram shows an example including an abstract dependency from the core schema, the specialization to the abstract class of a functional dependency, and finally to a boot dependency in the customized model. The boot dependency describes that a terminal is dependent on a DHCP server at boot time. Instantiation of the classes of the customized model produces the instance model which represents the managed system at runtime.

Leveraging model derivation. The models on adjacent levels are related to each other in that the more concrete model is derived from the more abstract, shown with grayed lines in the diagram. Two different derivation alternatives are employed: inheritance and instantiation, both imparting features to the more concrete model.

This circumstance can be exploited to minimize the effort needed in conflict detection, since any aspect of an abstract model will be present in the more concrete ones. Rightmost in Fig. 2 the activities described in the following sections of this paper are mapped to model abstraction levels. Note that derivation of invariants and conflict definitions (done manually) are performed on the abstract levels, where model size is small or moderate. In contrast, automated conflict handling resides on the most concrete model.

3 Using models to support conflict handling

Setting for policy-based management. Policies that are to be evaluated concurrently (e.g. because they triggered on the same event) are said to compose a *situation*. This situation may contain a conflict or not. The detection scheme presented in this paper determines the existence of a conflict and identifies the conflicting policies based on a frequently encountered situation definition. The last section hints at how to redefine the situation in order to extend or adapt the conflict handling approach.

Informal conflict notion Before detailing the application of models, we need to differentiate actual conflicts from other misbehavior of a policy system. The following three common conditions must be satisfied for a policy conflict to be possible ([13])

- Conflicts occur between two or more policies.
- Policies are evaluated concurrently.
- Goals of policies cannot jointly be reached.

In this paper's perspective, these conditions can be summarised as:
• *Constraints extracted from* a priori *models must not be violated by policies in the same* situation. Though we rule out ´conflicts´ occurring from the execution of a single policy, the model based approach could be used to detect such faulty policies as well.

To clarify the settings assumed for a policy-based management scenario, Fig. 3 shows three abstraction planes relevant to the approach presented in this paper. The management plane at the top includes the set of policies, where decisions are made and the execution of operations is initiated.

In the model plane, which corresponds to the instance model in Fig. 2, an implementation view is assumed in addition to the model view depicted. Thus, the MOs also imply agents, specifically policy execution points (PEP) able to enforce policies by executing actions on the underlying infrastructure. From the model perspective, the MOs hold a representation of an element while from the implementation perspective they constitute a middleware layer, obscuring the heterogeneity of the infrastructure. To avoid crowding the diagram, these two perspectives are not differentiated between. Finally, the infrastructure plane at the bottom includes all resources (hardware, applications, services etc.) to be managed.

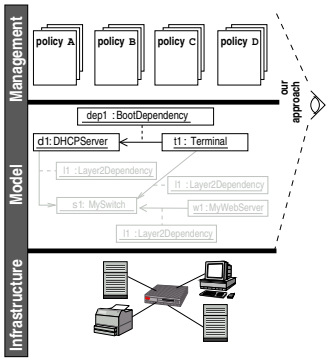

Fig. 3. Model and Views

Required Policy Components Policies can be specified at different abstraction layers, ranging from corporate or high-level policies at an abstract level, down to operational policies at a technical level. Though the methodology presented in this paper may work at a higher level of abstraction, it is targeted at the operational policy level. It is a requirement that policies be expressed in a formal policy language, rather than in prose.

Policy languages provide different sets of language elements. While the conflict detection methodology is applicable to any language, the latter must provide a minimum expressiveness (see also Fig. 1):

event The concept of a *situation* mentioned in Section 3 implies that the language have an event clause. (Examples are time, alarm etc.)

target Since our approach focuses on models of the target MOs, we need a target clause stating the MOs (or management domains) which are to be manipulated.

action Finally, an action clause is necessary to determine the concrete action to be executed on target objects.

3.1 Approaching conflict formalisation

The example in Fig. 4 shows an obvious conflict that cannot be detected or resolved without knowledge from models. It serves as motivation for the approach proposed in this paper while indicating in general the information needed to tackle conflicts of this type. The two policies A and B shown in the figure manipulate DHCP servers resp. the terminals (more precisely: call operations on the managed object boundary of the MOs). The notation in the target field of policies describes a *domain* which is a set of MOs build along management aspects [11].

Using an approach that only considers the management plane from Fig. 3 (i.e. solely the policies themselves) the conflict will not be detected, since the policies address different objects in different domains. Moreover, the goals to enable and disable different MOs are not per se conflicting.

If we consider the management plane and the model plane *in addition* (see Fig. 3), we can determine that the terminal `t1` has a *boot dependency* (which is a kind of a *functional dependency*, see Fig. 2) on the DHCP server `d1`. This circumstance is reflected in the instance model (Fig. 2). To achieve the goal of policy A (the dependent terminal can be used), the DHCP server also has be to usable. Policy B prohibits this goal by disabling the DHCP server. With the dependency information gathered from the model it becomes obvious that achieving the goals of both policies at the same time is not possible. Thus, a conflict between policies A and B has been detected.

```
policy { id="A"
  event { 8 am }
  target { /terminals }
  action { enable() }
}
```

```
policy { id="B"
  event { 8 am }
  target { /DHCP }
  action { disable() ;
           update(secPatch)}
}
```

Fig. 4. Example of conflicting policies: Is there a conflict between policies A and B?

3.2 Invariants of managed systems

Every managed system is governed by implicit rules resulting from its design. They range from very simple ones (e.g. a unit cannot perform its tasks while switched off) to complex dependencies between components or services. Invariants that formally describe these rules can be extracted from the models of the managed system. Again, different types of models will yield invariants of a different perspective.

A similar concept is found in the integrity constraints common in relational database management systems (RDBMS). An example for their purpose is to ascertain that a data set is not deleted as long as a dependent data set exists. Attempted violations of these constraints are interdicted by the DBMS.

The realization of this concept is eased by the fact that the number of actions is small (for an SQL-DBMS: insert, update, drop ...) as is the number of different data structures (relation, set/row etc). The concise action set found e.g. in DBMS is the result of well-adopted standardization. In systems and service management, the number of available management actions as well as their semantics is far from uniform.

Invariants are indicators for conflicts. A management action resulting in the violation of an invariant suggests a management problem, since with it an intrinsic rule of the system has been broken. A conflict between actions is indicated when the combined execution of two or more actions in the same situation results in the breach of an invariant.

4 Conflict detection: Step by step to conflict definition

In this section we outline the methodology for extracting conflict definitions and detection clauses from models. In the following, *conflict definition* refers to specific kinds or classes of conflicts (as in Fig. 8), not the generic policy conflict as such. A conflict is defined by stating model inherent requirements that are violated when a conflict occurs. Below, we present the steps necessary to distill such conflict definitions based on models at a high abstraction level. These definitions are still valid in the more concrete, derived layers of the model hierarchy, thus reducing the effort for conflict definition.

Step 1: Select a type of model. Any one of the available models can be selected in this step and it makes sense to perform these steps for more than one model. To

illustrate the principle by example, we will demonstrate the methodology for functional dependencies.

Example Selecting functional dependencies as the focused type of association, the level of abstraction to define invariants has to be chosen. To reach optimal reusability of definitions we chose the highest level of abstraction — here it is the level of abstract classes. Fig. 5 shows a section of a object oriented class hierarchy which might be derived from the generic CIM schema.

An MO `StateManagedObject` has possible states `Disabled` and `Enabled` and two methods to change the state. `Enabled` means that the resource is ready to execute user requests while `Disabled` means that usage is prohibited. A `StateManagedObject` can be linked with another `StateManagedObject` by an association called `FunctionalDependency`. Thereby, an MO can have the role of a (functional) dependent or an MO provides functionality (antecedent). The exact definition of this association class is made in the next step.

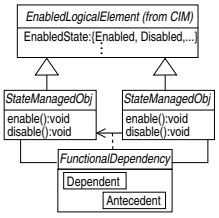

Fig. 5. Detailed customized model of functional dependency

Step 2: Extract invariants from the model. An invariant describes a model aspect in a formal, machine processable way and it can be evaluated to boolean values. Invariants can be classified into general invariants and specific ones. An example for a general invariant of a containment relationship is: the enclosing managed object (MO) must exist at least as long as the contained MOs. Such an invariant is inherent to a model; it is not related to the policies specified for the system. In order to specify invariants, an appropriate language has to be chosen.

Object Constraint Language (OCL) The Unified Modeling Language (UML) defines a formal language to describe constraints for any (UML) model. OCL [12] can be used to define invariants, pre- and postconditions as necessary for our methodology. For our purpose, all invariants are defined for classes and must hold for all instances of that class. Though any equivalent formalisms can be used instead, UML does provide a common language for graphical presentation of object models and OCL offers the opportunity of using an OCL compiler to translate invariants (and consequently conflict definitions) to executable code, thus enabling their direct use.

Example An invariant consists of two parts: the `context` part which describes the starting point of the invariant (here it is the class `FunctionalDependency` from Fig. 5) and the `inv` part which contains the constraint. To specify an invariant for functional dependencies, we have to reflect which condition has to hold (is always evaluated to ´true´) for the whole life time of that association: the invariant in Fig. 6 states that if an instance exists in the `dependent` role and its status is `Enabled` then the antecedent has also to be in its `Enabled` state to ensure proper execution of the dependent. (The authors are aware that this is only one possible definition out of a huge set. As the paper focus on conflicts we leave a discussion of optimal dependency definition.)

```
context FunctionalDependency
inv     funcdep:
        self.Dependent → exists (mo | mo.status='Enabled')
        implies self.Antecedent.status='Enabled'
```

Fig. 6. Invariant 1

The keyword `self` refers to an instance of the class `FunctionalDependency`. With help of a dot you can navigate through the model: `self.Dependent` is the set of instances of the class `StateManagedObject` which hold the dependent role in this concrete instance (`self`) of `FunctionalDependency`.

Step 3: Derive relationships of invariants to policies. In this step, invariants are mapped to policy actions. For this purpose, the general invariants mentioned in the previous step are considered along with the policies.

While policies contain actions to be executed, invariants do not. Therefore, the effect of the actions on the model needs to be specified. As shown in the example in step 2, all actions changing the state of the associated MOs are described there by describing the effect of an MO´s method as postconditions.

Postcondition 1

context StateManagedObject :: disable ( ) : void
post: self.status = 'Disabled'

Postcondition 2

context StateManagedObject :: enable ( ) : void
post: self.status = 'Enabled'

Fig. 7. Postconditions

Example The abstract class `StateManagedObject` consists of two methods `disable()` and `enable()` which are described in OCL (Fig. 7). Postcondition 1 states, that after termination of the method the attribute of the class `StateManagedObject` has the value `Disabled`. Postcondition 2 is defined in the same way.

Step 4: Create conflict definition. The conflict definition "parts" determined in the preceding steps are combined in this step. A conflict definition describes the circumstances in which conflict occurence is certain. Having defined an invariant (step 2) and post conditions for the methods (step 3) this step analyzes if the invariant can be evaluated to ´false´ and if so, a conflict definition is generated.

Example As the functional dependence has two different ends (dependent and antecedent) and the objects associated have two different methods (`disable()` and `enable()`) there are 4 pairs of methods call to examine (Fig. 8).

For the first pair, the invariant cannot be evaluated to ´false´ in any execution order, hence this pair is always conflict free. The next two pairs are in conflict depending on the execution order (and are therefore race conditions). Conflict definition 1 and 2 in Fig. 8 reflects this situation. The last pair is in conflict disregarding the execution order (see Conflict 3).

Conflict 1

conflict FuncDepDisable
context FunctionalDependency
conflict space: MO[mo1] :: disable( )
MO[mo2] :: disable( )
pre: self.Antecedent = mo1
and self.Dependent = mo2
refers to: inv FunctionalDependency

Conflict 3

conflict FuncDep
context FunctionalDependency
conflict space: MO[mo1] :: disable( )
MO[mo2] :: enable( )
pre: self.Antecedent = mo1
and self.Dependent = mo2
refers to: inv FunctionalDependency

Conflict 2

conflict FuncDepEnable
context FunctionalDependency
conflict space: MO[mo1] :: enable( )
MO[mo2] :: enable( )
pre: self.Antecedent = mo1
and self.Dependent = mo2
refers to: inv FunctionalDependency

Method pairing

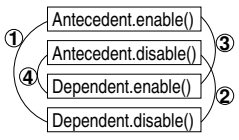

Fig. 8. Conflict definitions and method pairs

To describe a conflict definition OCL is extended by the following keywords: **conflict** to denote the name of the conflict, **conflict space** to identify the actions/operations relevant to the conflict and **refers to** to identify the invariant that the conflict references.

As the execution of operations always is the cause conflicts, naming the involved operations is an important part of the definition. The optional precondition narrows the context of the operations when a conflict occurs.

Step 5: Provide conflict detection clause. For a conflict definition, a detection clause is sought that accounts for the characteristics specific to the conflict definition. In our example, if a `StateManagedObject` is in the role of the antecedent , and wants to `disable()` then it must be ensured that all dependents are already in the state ´Disabled´ (see Precondition 1 in Fig. 9).

Precondition 1

context MO :: disable( ) : void
pre: self.Antecedent
implies
self.Antecedent.Dependent.forall(
mo | mo.status = 'Disabled')

Precondition 2

context MO :: enable( ) : void
pre: self.Antecedent
implies
self.Antecedent.Dependent.forall(
mo | mo.status = 'Enabled')

Fig. 9. Preconditions

With respect to this precondition, a sequential execution of policies could resolve a conflict.

5 Conflict handling

In this section we show how the concepts developed in Sections 3 and 4 can be applied to detect and handle policy conflicts. During operation of the managed system, policies are triggered by events generated in the system. Before their actions are executed, the set of policies in a situation (see Section 3) are analysed with respect to possible conflicts. If a conflict is detected, the strategies presented in Section 5.2 can be applied to attempt its resolution.

5.1 Application of conflict detection

Fig. 12 shows an activity diagram of the algorithm for conflict detection and resolution. Since the algorithm works on sets of policies, it depicts such sets (instead of objects) as input and output of the activities.

To illustrate the conflict handling algorithm, we use the situation shown in Fig. 10 as an example. In addition to the two policies (A and B) from Fig. 4, two other policies are triggered by the same event: one that specifies that the webserver should reread its configuration files (C), and one that enables the printing service (D).

In an example using as few as four policies, some of the steps described in the following may seem redundant. When considering a large number of policies operating on large models, these steps ensure that all situations are handled correctly.

```
policy { id="A"
  event { 8 am }
  target { /terminals }
  action { enable() }
}
```

```
policy { id="B"
  event { 8 am }
  target { /DHCP }
  action { disable() ;
           update(secPatch)}
}
```

```
policy { id="C"
  event { 8 am }
  target { /printer }
  action { enable() }
}
```

```
policy { id="D"
  event { 8 am }
  target { /webserver }
  action { rereadConfig() }
}
```

Fig. 10. Example *situation*

Prerequisites Three information sets are needed for conflict handling: policies in a situation, e.g. those shown in Fig. 10; conflict definitions, e.g. those described in Section 4; models of the system, as shown in the model plane of Fig. 3. These sets are related, as shown in Fig. 11: The

policies contain actions and references to MOs (targets). Actions relevant to policy conflicts are found in the conflict definitions, as are MOs and associations of MOs. In addition, the conflict definition specifies patterns of association between MOs in the models. To perform conflict handling, we leverage the relation between these information sets.

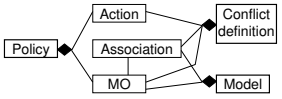

Fig. 11. Information sets

Match conflict definitions The policies in a situation are tested against the conflict definitions acquired by means of the procedure described in Section 4. Specifically, the actions of the policies are compared to the `conflict space` fields of conflict definitions. Matching policies constitute a *matching set* within the potentially conflicting set.

In our example situation, the actions of the policies A - D are matched to the actions of the conflict definitions 1-3 (Fig. 8). It is obvious that the policies containing methods `enable()` and `disable()` will match. Thus, the matching set contains policies A, B and D. These are used as input to the next activity.

Sort by conflict definition From the matching set, a number of sets $cd(i)$ are created, each corresponding to exactly one conflict definition i. The sets may overlap, since a single policy may match several conflict definitions.

In our example, the conflict space of all conflict definitions contain the methods `enable()` and `disable()`. Hence, all the policies in the matching set match all conflict definitions, so that three sets $cd(1)$, $cd(2)$ and $cd(3)$ result, each one of them containing policies A, B, D. Thus: $cd(1) = cd(2) = cd(3) = \{A, B, D\}$

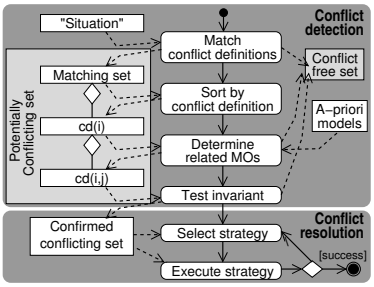

Fig. 12. Conflict handling algorithm

Determine related MOs The previous activity has correlated the policy actions and the conflict definitions, thus identifying potentially conflicting policy sets. To violate an invariant, actions must be executed on objects that are related according to the invariant. To test this condition, we compare the targets of every policy in every set $cd(i)$ with MOs referenced in the conflict definitions and determine whether the roles they carry in the model (e.g. a dependency) corresponds. Again, the policies in a set $cd(i)$ can be related to several model partitions, and several roles j may have to be tested. All possible combinations of policies from a set $cd(i)$ result in sets $cd(i, j)$. As in the previous step, the resulting $cd(i, j)$ may overlap.

To clarify this step, consider the following procedure:

1. Select a set to begin with, e.g. $cd(1)$
2. Find all instances j of the class from the invariant of i: results in all instances of `FunctionalDependency`.
3. Determine targets of policies in the set $cd(1)$ selected: result is `terminal`, `DHCPserv` and `printer`
4. For all j, find instances containing the targets above: results in sets $cd(1, j)$, where j is the current instance of i's invariant's class.
5. Repeat the steps for all remaining $cd(i)$.

The example model only has one association that is relevant for conflict detection, thus the only set yielded by this procedure is $cd(3, 1)$: it matches the pattern Antecedent.disable/Dependent.enable. Since there is no dependency between the `printer` MO and the other two MOs, only policies A and B are left in the set.

At this point, both actions and targets have been accounted for. Further analysis of a set $cd(i, j)$ can be performed disregarding the other sets.

Test invariant The previous activity has created sets that correspond to single conflict definitions and contain only policies with a high potential of conflict. This activity corresponds to a simulation of the execution of the policy actions. It divides the sets yielded by the previous step into a *confirmed conflicting set* of policies and conflict-free policies. For each policy set $cd(i, j)$, the invariant referenced in the conflict definition i is tested and evaluated under the assumption that the actions of the policies in the set are executed. Since policies are executed in parallel, the invariant must be tested for all permutations of the serialization of the set. If the set passes all tests, the policies in the set are released from the potentially conflicting set into the conflict-free set. If the invariant evaluates to *false* for at least one permutation, the set is conflicting.

In our example, the terminal is dependent on the DHCP server in that that service must be available for the terminal to boot. The invariant of conflict definition 3 is violated, since the object in the dependent role (terminal) is enabled, while the object in the antecedent role (DHCP server) is disabled. In consequence, the set $cd(3, 1)$ is transferred to the confirmed conflicting set.

Select/execute strategy For each conflicting $cd(i, j)$, a resolution strategy must be selected and executed. As shown in Fig. 12, more than one of the strategies discussed in Section 5.2 may be tested, resorting to Strategy A if all others failed. An optimal selection of strategy depends on the number of policies in a set, how time-critical their execution might be and possibly other factors not taken into consideration yet. The effectiveness of the resolution strategy can be tested by applying the same steps as for conflict detection to a modified set of policies.

For our example, a sequential execution of the conflicting policies in appropriate order resolves the conflict (let the terminal boot before disabling the DHCP server).

5.2 Strategies for conflict resolution

As discussed in Section 5.1, conflict detection determines a policy set where two or more policies are in conflict with each other. Once this set is known, attempts can be made to resolve the conflicts in an automated fashion, or at least minimize their impact. This section presents strategies to that end.

Existing strategies exhibit an all-out approach to conflict resolution, as the following two alternatives show:

A The most drastic measure is for the policy service to abstain from enforcing *any* of the policies. This could prove to be a viable approach in applications that are not time critical, but it still requires manual intervention.

B Another approach found in the literature ([5,8]) is to enforce only the policy with the highest priority in the set – assuming policies have been assigned priorities. This scheme is often mentioned in the context of quality of service policy, and its usefulness may be constrained to that niche.

Strategies using the automated conflict detection presented in this paper allow for a differentiated resolution. Since conflicts can be detected in an automated manner in any set of policies, combinations of the policies in the set can be tried. This yields the following strategies:

C Try to enforce as many policies as possible, excluding the minimal number of policies for the set to be conflict free. We can determine the set to be enforced by iteratively applying conflict detection to parts of the conflict set.

D Serialising parts of the policy set and finding an appropriate synchronous enforcement order can resolve the conflict in some cases. This solution appears to be the least invasive, though it will slow down the enforcement of the policies. Again, the enforcement order is found by applying the conflict detection algorithm to permutations of the conflict set.

E Create conflict free subsets and serialise the enforcement of subsets. This is an optimisation of the above strategy. While the policy subsets are enforced synchronously, the enforcement of the policies in one subset can be parallelised. Normally, i.e. in cases where the number of sets is small compared to the mean number of policies per set, this strategy should execute faster.

These alternatives are orthogonal: any combination of serialisation and reduction of the policy set is possible. Unfortunately, seeking the optimal solution implies testing a large number of policy sets, which may not be practical to do at runtime.

6 Related work

Policy conflict handling We imposed the requirement that the conflict handling scheme presented in this paper be independent of a specific policy language, the type of policy

and overlap of policy domains as a necessary prerequisite. In the following, we present work related to the approach presented here.

At Imperial College much valuable work in the area of policy conflicts has been published. The result of [10] is a conflict classification. Conflicts are classified along the number of overlaps (at least one) of domains (given in subject, target or action) between two or more policies.

Another conflict detection approach exploiting domain overlapping is found in [8]. It focuses on modality conflicts, where policies are typed (positive and negative authorisation and obligation policy). A triple overlap (subject and target and action) indicates a conflict. This approach is effective in detecting conflicts between authorisation and obligation policies, but limited regarding conflicts between obligation policies.

Damianou [5] uses so called meta policies, which are part of the Ponder specification, to formalize policy conflicts. A meta policy specifies constraints regarding a set of policies. As this approach is language specific and not all policy languages support the concept of constraint-based meta policies, the general applicability is limited. Also, to apply the general-purpose tool of meta policies to conflict handling, a methodology for the specification of appropriate meta policies would have to be created.

The scope of [1] is to support policy refinement. A formal language (event calculus) is used to represent the state of a system allow reasoning about possible future states. Policy language, policy execution and the managed system are formalised using event calculus. Based on the resulting model, conflicts can be defined in event calculus, overlap of domains being a prerequisite for conflict definition.

This approach takes into account the managed system allowing calculus representation of static and dynamic aspects of a whole managed system. However, since since special models must be created, the effort introduced seems to be quite high, especially when considering large scale, complex systems.

In [2,3] a policy conflict resolution approach is shown for the Policy Description Language $\mathcal{PDL}$, a rule-based language which omits the policy element subject and target. Conflicts are defined by monitors which evaluate action constraints. An action constraint has the form: **never** `Action1` $\wedge$`...`$\wedge$ `ActionN` $\wedge$ `condition` . A methodology to derive action constraints is not given, also the application of the approach to other policy languages and architectures is not discussed.

Management object modeling To be able to derive conflict definitions we need the concept of object orientation especially the concept of classes and methods. Management classes provide abstraction of resources for management purposes.

The instances of a management class and their embedding in a management information base (MIB) is standardised. Well known object oriented MIBs are DMTF's Common Information Model (CIM)[4] and ISO's Structure of Management Information (SMI) [6,7]. For these standards our approach can be applied directly.

For IETF's SNMP-SMI [9] (also known as Internet MIB), which is not object oriented, a wrapper needs to be designed in order to allow our approach to be applied to MIB attributes.

7 Conclusions

In this paper we have discussed an approach to conflict handling relying on a priori models. Different types of models represent static and dynamic aspects of managed

systems. They can be leveraged to derive invariants that, in turn, yield conflict definitions. Aided by these conflict definitions, policy sets can be checked for conflicts either statically or at runtime. In the following we summarize the key concepts of the paper and point to further topics of study.

We presented a methodology to derive conflict definitions from object oriented management models. In addition, we have shown how to perform automated conflict detection based on these conflict definitions. Also, we presented conflict resolution strategies for the policy sets found to be in conflict. The approach presented is generic in that it has no dependency regarding type of policy, the policy language used or management model type. Merely three policy elements are prerequisite to using the approach.

Yet, several important topics of study in the area of conflict handling remain. An important issue would be the integration of conflict handling methods for different types of policies, e.g. approaches targeting authorisation policies using a model based scheme.

We assumed that a *situation* consists of policies that have been triggered by one event. Keeping in mind that a situation is merely a set of policies, the same concepts can be extended to support policy sets created in other ways, e.g. by observing a sequence of events. For that purpose, the only thing that needs to be changed is the definition of the situation itself. As a related issue, using invariants to identify faulty policy specification seems to be a rewarding topic.

References

1. Arosha K. Bandara, Emil C. Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *Proceedings of HPOVUA 2003*, 2003.
2. J. Chomicki, J. Lobo, and S. Naqvi. A logic programming approach to conflict resolution in policy management. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000)*, pages 121–132, Breckenridge, Colorado, Morgan Kaufman, 2000.
3. J. Chomicki, J. Lobo, and S. Naqvi. Conflict resolution using logic programming. *Transaction on Knowledge and Data Engineering*, 15(1):244–249, 2003.
4. Common Information Model (CIM) Specification Version 2.8. Specification, January 2004.
5. N. C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, February 2002.
6. Information Technology – Open Systems Interconnection – Structure of Management Information – Part 4: Guidelines for the Definition of Managed Objects. IS 10165-4, International Organization for Standardization and International Electrotechnical Committee, 1992.
7. Information Technology – Open Systems Interconnection – Structure of Management Information – Part 7: General Relationship Model. IS 10165-7, International Organization for Standardization and International Electrotechnical Committee, 1997.
8. Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November 1999.
9. K. McCloghrie and M.T. Rose. RFC 1065: Structure and identification of management information fo r tcp/ip-based internets. RFC, Internet Engineering Task Force (IETF), August 1988.
10. Jonathan D. Moffett and Morris S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 1993.
11. Morris S. Sloman and Kevin Twidle. *Domains: A Framework for Structuring Management Policy*, chapter 16. 1994.
12. OMG Unified Modeling Language Specification, Version 1.5. Technical Report formal/03-03-01, Object Management Group, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.
13. A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. RFC 3198: Terminology for policy-based management. RFC, Internet Engineering Task Force (IETF), November 2001.