Intelligent Assistant for Context-Aware Policies

Helen Balinsky and Neil C. A. Moore
Hewlett Packard Laboratories,
Bristol, UK
helen.balinsky@hp.com & neil@bigoh.co.uk

Steven J. Simske
Hewlett Packard Laboratories,
Ft. Collins, CO, USA
steven.simske@hp.com

***Abstract*—Recent advancements in document handling tools have greatly simplified the tasks of reusing content and copying between secure and unsecure locations. As a result, intentional and unintentional data leaks are occurring more often and so posing a serious risk. To address this issue, context-aware policies have been introduced so that the policies pertain to the patterns of text contained in actual run-time content – as well as static document-level metadata – with the aim of distinguishing different types of sensitive documents. Unlike traditional metadata based policies, context-aware policies do not need to have any overlap in their conditions for contradictions to occur, which makes it harder to create and manage coherent, ambiguity free sets of policies. We provide a novel Intelligent Modeling Assistant (IMA) that automatically deals with resolvable complications in modelling, while breaking tasks requiring manual intervention into a set of simple decisions. The IMA also helps the administrator to understand possible consequences of policies through sets of examples. Our implementation using constraint satisfaction technology is described.**

I. Introduction

The last decade has brought unparalleled advances in document creation and management technologies: from collaborative mash-ups to automatic repurposing tools, and from document centric workflows to on-line document sharing. Cloud computing and mobility have merged secure intranet and insecure internet making it simple to drag-and-drop protected data into a public document, often even without realizing it. That is, document level metadata – the cornerstone of traditional document access control – is insufficient when copied data is leaked. Such metadata could easily be lost or fail to properly describe a newly created mash-up.

Context aware policies are a new technique for policy enforcement. They take into account the actual (run-time) document contents at the moment a document action is about to be executed. An example of such a system can be found in [1]. Policy conditions of context-aware policies may include document keywords, data patterns, regular expressions or any combination thereof. When a document being exported is scanned and the policy condition is satisfied, then the protective action defined by the policy is triggered thus preventing an inadvertent sensitive data leak.

It is expected that documents from the same office have a lot in common, relating to the same subjects and topics. Yet, only some of them are sensitive and must be distinguished by policy conditions. In addition, in natural language there are many ways to express the same things, thus requiring for a policy to be flexible enough to accommodate potential variations as well as language inflexions. Hence in order to be descriptive enough, context aware policy conditions might need to incorporate alternatives, negations and variants.

In §II and §III of this paper we describe the problem and give an overview of our solution, and in §IV describe related work. In the remaining sections we describe our solution in detail and finish with concluding remarks.

II. Problem Statement

A first problem is how to make decisions when end users attempt to export data from their systems. A solution must evaluate the policies quickly and accurately, so that the user is not held up in their work but sensitive data cannot be leaked.

The next issue is how to create and define a set of policies. As well as manually creating new policies, the administrator should be aided in understanding other policies created automatically. Mixing complex hand-made and automatic policies with elaborate policy conditions while keeping the overall policy system comprehensive and consistent, presents a substantial technical and usability challenge for our assistant.

A further complication presents itself in the management of ambiguities in policies, where multiple policies apply to the same document. We will help the administrator to specify the intended behaviour in the event of a ambiguities by presenting them with a sequence of choices based on example documents. As we will show in §IV, ours is the first technique that we know of that helps the adminstrator in this way rather than simply asking them to manually amend the set of policies to avoid conflicts.

The latter two difficulties in modelling motivates our work in this paper on an Intelligent Policy Assistant for creating and managing next-generation context-aware policies, whilst automatically keeping them free of inconsistencies, redundancies and contradictions.

III. Our Solution in Brief

We will show how to solve the following problems in the context of our Automatic Policy Enforcement eXchange (APEX) solution [1], [2] for data leak prevention. APEX, a policy enforcement solution, captures full document content and document metadata before potentially policy-breaching action has been fully executed and sensitive data has escaped. Thus, such action is instantly suspended and the document data and metadata are analyzed using context-aware policies to determine whether the required action can continue or, if

not, what the alternative action should be. As with any real run-time system, the decision should be made quickly: a user is waiting for the operation to be finished. Our solution provides a mechanism for making such policy decisions efficiently. We implement this functionality using constraint programming and specifically a satisfiability solver.

The second part of our solution is to create an assisted editor for setting and maintaining security policies. The object of this is to move beyond simply specifying the policies and observing their effects on a working system, and instead to create an editing environment that helps users to understand and predict the effects of their policies during editing. The aim is to avoid any unintended side effects of any policy–wasting the user's time, allowing unsafe actions, and/or creating a security hole–while also ensuring that no safe actions are inadvertently denied. We also aim to avoid policy administrators being exposed to the modelling formalism directly, but instead guide them with feedback and assistance based on examples of documents and actions.

IV. Related work

We believe that our line of research (previously published in [1] and [2]) is unique in that the policies are based on runtime document contents as well as metadata. This makes our policies different from many types of policies used in the past because they may pertain to a boundless set of attributes, e.g. the presence or absence of an arbitrary word in a document, and not a predefined set of fields as in the past.

We take care in this paper to explain how we deal with *conflicts*, where multiple policies contradict each other on certain attempted actions. There is much previous work in this area. Numerous papers have given definitions of types of conflicts, e.g. [3]–[10]. The type of conflicts we are interested in are called *modality conflicts* in [4]. There are several existing techniques for handling conflicts. Explicit prioritisation of policies is found in [4], [7], [11]–[13]. Our solution to this problem is by explicit computer-assisted prioritisation. Other ways to resolve conflict include prioritising restrictive (or permissive) policies ([11], [13]); domain-specific techniques that prioritise the most specific policies ([3], [5], [14]); prioritising the most recent policies [14]; or simply to forbid conflicts ([3], [13]) (usually as a last resort once other techniques fail). Numerous approaches exist to discover conflicts in sets of policies ([4], [7], [12], [15]–[19]). Our technique is most similar to [6] in spirit. This technique processes policies in order to determine when pairs are such that one makes the other redundant, one a special case of the other, etc. In [6], the conflicts are reported to the user, whereas we incorporate this type of reasoning into an assisted wizard to help the user to obtain their desired behaviour for the whole set of policies.

Our work also focusses on intelligent interfaces for editing and understanding sets of policies. Interfaces which assist the user in finding conflicts have been implemented before and described in [17], [19]–[21]. These interfaces all assume a high level of administrator competence; in particular they assume that the administrator is conversant in the modelling formalism and that it is enough to list the conflicts for him to figure out how to change his set of policies to deal with them. In our approach, we assist the user in making simple choices between example documents, rather than asking them to directly edit policies or specify priority between policies.

Our system is implemented using constraint programming and satisfiability. Constraints have been used before to model policies in [22] and [23], though the details of our modelling differs. Logic programming ([18], [24], [25]) and formal models have also been used in the past for encoding policies ([15], [21], [26], [27]).

V. Policies

A. Policy system

In this section we will formally describe the format of policies and what they mean. Each policy is as follows:

$$\begin{aligned} rule ::= & proposed\_action \\ & \wedge\ metadata \\ & \wedge\ policy\_expr \rightarrow required\_protection \end{aligned}$$

where

$$proposed\_action ::= print|email|upload|save,$$

$$\begin{aligned} metadata ::= & printer\_IP|email\_address \\ & |\ upload\_IP|save\_path, \end{aligned}$$

$$\begin{aligned} policy\_expr ::= & policy\_condition \\ & |\ (policy\_expr \vee policy\_expr) \\ & |\ (policy\_expr \wedge policy\_expr) \\ & |\ (\neg policy\_expr) \end{aligned}$$

$$policy\_condition ::= text\_tag|regular\_expression$$

$$\begin{aligned} required\_protection ::= & allow\ [allow\_embellishment] \\ & |\ deny\ [deny\_embellishment] \end{aligned}$$

$$allow\_embellishment ::= log|encrypt|sign|redact|\ldots$$

and

$$deny\_embellishment ::= log|alert|\ldots.$$

In valid rules the metadata must match the proposed action, e.g. for printing the metadata must be a printer IP address, and the policy conditions should be respectively strings of one or more characters and valid regular expressions.

A $text\_tag$ or $regular\_expression$ evaluates to $true$ when it is found anywhere in the document. The meaning of $policy\_exprs$ correspond to normal meanings in boolean algebra. Policy rules correspond to their boolean algebra equivalent; that is, whenever $proposed\_action$ and $metadata$ match the action the user is trying to carry out and $policy\_expr$ evaluates to $true$ on the document that the action pertains to, then the specified $required\_protection$ is forced on the action.

Required protections fall into two classes, $allow$ and $deny$, but may also have an embellishment, which means that the protection must be applied along with an additional feature. For example, $allow\ encrypt$ means that the action should be allowed combined with encrypted. We say that two policies have *compatible* protections when they are the same apart from embellishments. We say that two policies have *incompatible* protections when they are different apart from embellishments.

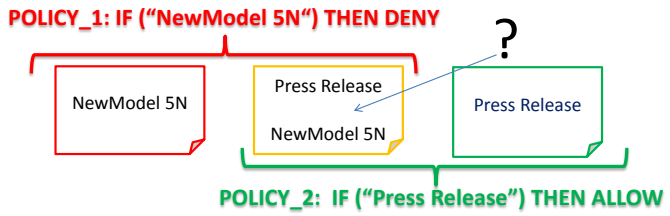


Fig. 1. Depiction of policy contradiction

Example 1: Consider the following example policy

$$save \wedge \neg \texttt{'C:\encrypted'} \wedge \texttt{'classified'} \rightarrow deny.$$

This policy will be in effect only for actions that intend to save a document containing the word "classified", outside the `'C:\encrypted'` directory path, and it says that action should be denied. For any other action the policy will be ignored as it does not apply. ■

It is a common situation for multiple context-aware policies to apply at once. Note that in a policy language with fixed condition attributes, e.g. source and destination IP address and port for firewall rules, only policies that overlap on their attributes can possibly apply simultaneously and therefore possibly contradict each other. On the other hand, *our policies do not need to have any overlap in their conditions for contradictions to occur.*

Example 2: Suppose we want to model policies so that the user is forbidden to email any file containing the name of a new product (called NewModel 5N). However documents containing the words "press release" should be allowed. We want these policies to avoid leakage of documents related to the new product, except explicitly vetted press releases. A candidate set of policies are depicted in Figure 1. When a document contains both "press release" and "NewModel 5N" then we have a policy contradiction. Note that the policies do not overlap in their conditions. ■

B. Interpretation of policies

§V-A describes the meaning of a single policy, however we are interested in compiling databases of policies to describe overall security policy. Hence we need to produce a policy decision when a *set* of policies is applied. On an intended action when only one policy applies, the rest are simply irrelevant. When two policies suggest the same protection for a particular action, they can both have their way. However when two policies suggest incompatible protections applied to the same document, e.g. $allow$ and $deny$, we need a way to decide which policy decides the final outcome. We call the situation where multiple policies have incompatible outcomes *policy contradiction.*

Before talking about contradiction in detail, we will describe what happens when multiple policies suggest compatible but different protections, e.g. $allow\ sign$ and $allow\ encrypt$. All the embellishments that we will consider are what we will call *stackable*, meaning that they can all be done at once, i.e. it is perfectly acceptable to sign, encrypt and log the same document. Hence when multiple policies apply we just do all the embellishments in some predefined order.

We will consider 3 strategies for handling contradiction:

Most restrictive action If multiple policies return different results, use the most restrictive result, e.g. $deny$ when any rule says $deny$. Note that *least* restrictive is also a valid strategy.

Normalisation Ensure that one and only one policy applies at any time by constructing the policies carefully.

Prioritisation Order the policies and use the result of the highest priority one that applies.

Each strategy provides a consistent and precise way of dealing with policy contradiction. The first choice, most (least) restrictive action, is inflexible because it has an intrinsic bias towards restrictiveness (permissiveness). The second choice, normalisation, is tricky to work with because policy conditions will have to be more specific, so that only one can ever apply at once. This is difficult for the average user to achieve, and even if the system were able to automatically normalise policies, the user may lose their intuition for their own transformed policies. As we will show, the remaining third choice, prioritisation, provides ease of policy modelling and flexibility to create policies for special cases. Similar strategies are described for XACML policy-combining algorithms (see [28], page 15). Using an example, we will now illustrate the strengths and weaknesses of each interpretation of policy contradiction:

Example 3: Consider the same scenario described in Example 2.

Using "most restrictive action" the policies needed are

$$email \wedge \texttt{'NewModel 5N'} \wedge \neg \texttt{'press release'} \rightarrow deny \quad (1)$$
$$email \wedge \texttt{'press release'} \rightarrow allow. \quad (2)$$

The things to note are that the policy denying leakage of details of "NewModel 5N" needs to specify explicitly that press releases are excluded. The policy for "press releases" is as concise and clear as we could have hoped.

Using "normalisation" the policies needed are

$$email \wedge \texttt{'NewModel 5N'} \wedge \neg \texttt{'press release'} \rightarrow deny \quad (3)$$
$$email \wedge \texttt{'NewModel 5N'} \wedge \texttt{'press release'} \rightarrow allow. \quad (4)$$

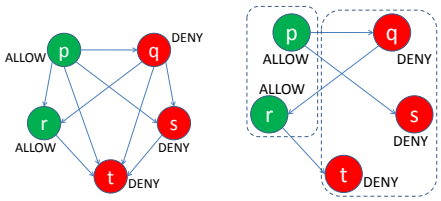
As can be seen from this example, normalisation results in very verbose policies, since they need to be specific enough that only one applies at once. For each additional product name that is to be protected, we must add two policies.

Finally using "prioritisation" we need policies

$$email \wedge \texttt{'NewModel 5N'} \rightarrow deny \quad (5)$$
$$email \wedge \texttt{'press release'} \rightarrow allow. \quad (6)$$

such that the latter policy has a *higher* priority than the former. These policies are quite clear, although some complexity is hidden in the ordering of the policies. The intuition behind the ordering is that the "press release" rule is a special case that overrides all other policies and is placed top priority.

For each alternative interpretation we will now describe which policies apply for a couple of example documents. For a document containing "NewModel 5N" and not "press



(a) Totally ordered (b) Partially ordered

Fig. 2. Priority relations drawn as graphs: vertices are policies and edges go from higher to lower priority policies

release" policies (1), (3) and (5) apply. Hence for each interpretation a single *deny* rule applies and the result is clear. However for a document containing both "NewModel 5N" and "press release" policies (2), (4), (5) and (6) apply. For "most restrictive" and "normalisation" only a single *allow* policy applies so there is no contradiction. For "prioritisation" both policies are in effect but the *allow* policy wins. ■

C. Avoiding contradiction

From now on we will assume that prioritisation is being used to avoid conflicts, but we have not yet specified precisely how the policies should be ordered. Orderings can be drawn as directed graphs as shown in Figure 2. The vertices in the graph are policies. If there exists a path from policy x to policy y then policy x has a higher priority than policy y. A first property required of the ordering is that for any pair of policies with incompatible protections, e.g. *allow* and *deny*, one of the policies must have priority over the other, i.e. $\forall x, y$ there is a path from x to y or a path from y to x, but not both. This means that whenever a pair of policies might conflict there is an unambiguous outcome. A concequence of this property is that the graph should not contain cycles because a cycle involving policies x and y is such that x is higher than y and also y is higher than x, meaning neither has priority.

In Figure 2(a) a complete ordering is shown, with edges between all pairs of policies, including those with the same protection. This corresponds to arranging policies in a list sorted from highest to lowest priority. However, it describes unnecessary ordering between policies; for example, two policies with protection *allow* are relatively ordered, but they do not conflict. Also, there is no need for the edge from p to t because there is already a path from p to t via r. It is important to minimise the number of edges because each edge corresponds to a decision the user has to make about policy ordering. For this reason we will avoid complete orderings.

In Figure 2(b) an ordering is shown that also satisfies the necessary properties but using far fewer edges. Now there is no ordering between some pairs of policies with the same required protection, e.g. policies q and s. Notice that the graph is *bipartite*, meaning that it can be split up into two sets of vertices such that all edges are *between* the sets.

Now we have described a suitable way of describing policy ordering, such that in any situation where multiple policies apply at once there is a well defined way to decide which result applies. Furthermore, the ordering is concise since fewer edges are needed.

In the following section we will describe how we implement these concepts in software.

VI. Constraint Programming Modelling of Policies

In the previous section we described in abstract terms what policies are and how they are evaluated to make policy decisions for data leak prevention. In this section we describe our constraint programming implementation of policies.

The constraint programming paradigm is based on a *modelling* stage where the problem domain is described in terms of constraints and variables and a *solving* stage where solutions are found. In our case the modelling is done using boolean satisfiability:

Definition 1 (Boolean satisfiability (SAT)): A SAT problem consists of

- a set of *variables* $V = \{v_1, \ldots, v_j\}$,
- a set of *literals* L each of which is either a variable v or its negation $\neg v$, and
- a set of *clauses* $C = \{c_1, \ldots, c_k\}$, where each clause c_i is a set of literals.

A *solution* to a SAT problem is a set of literals S such that $l \in S \Leftrightarrow \neg l \notin S$ and also for each clause C the intersection of C and L must be non-empty, i.e. a literal from the solution must be found in each clause.

A clause $\{l_1, \ldots, l_j\}$ behaves like a disjunction $l_1 \vee \ldots \vee l_j$ because the solution must contain at least one literal from each clause in order that it be *satisfied*. The whole SAT behaves like a conjunction $c_1 \wedge \ldots \wedge c_k$ because all clauses must be true for the SAT to be satisfied. When $v \in S$ we say that v is set to $true$ in the solution and when $\neg v \in S$ we say that v is set to $false$. ■

Example 4: Consider the SAT consisting of variables $\{x, y, z\}$ and clauses $\{\{x, \neg z\}, \{x, z\}, \{\neg y, z\}\}$. This corresponds to the Boolean expression $(x \vee \neg z) \wedge (x \vee z) \wedge (\neg y \vee z)$. The set $S = \{x, \neg y, z\}$ is a solution, because each clause has a literal from S in it. This corresponds to setting $x = true$, $y = false$ and $z = true$. ■

Hence, the modelling stage in our solution consists of generating a SAT problem that describes our security policy and the solving stage involves providing this model to a SAT solver. The attempted action is allowed under the policy if and only if the SAT solver can find a solution. When a SAT solver based on backtracking search terminates, it has either found a solution or proved that none exists. The best known lower bound for solving the SAT problem is exponential and indeed it was the first known NP-complete problem [29]. We rely on the fact that in practice solutions can be found quickly due the intelligence and efficiency of modern solvers and later in the paper we will provide experiments to justify this. Our implementation uses the SAT4J solver [30]. In the following section we will describe how we have modelled our security policies as a SAT.

A. Modelling of policy system

In this section we will describe the variables and clauses used to model sets of policies.

For the purposes of this model, notwithstanding the discussion in §V-C, we will assign each policy a priority number where higher numbers mean higher priority. This can be done by performing a preorder traversal on the priority graph, in order to create a list of policies from low priority to high and then numbering them in that order. Let the numbers assigned range between 1 and $maxprio$.

The previous section described a SAT problem as being defined in terms of clauses. To make the presentation simpler, in this section we will first describe the constraints on our model using boolean expressions involving conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and biconditional ($\leftrightarrow$). Following these will be an equivalence operator ($\equiv$) and then a concrete way of writing down the expression as a clause.

Each fragment of a policy (i.e. the parts of a policy excluding boolean operations) is given a boolean variable that is true if and only if the current document or proposed action matches it. For example there is a variable for each word appearing in a policy (e.g. "confidential") and a variable for each proposed action (e.g. "email"). Even if a fragment appears in multiple policies it has only one variable. For example the policy

$$email \wedge addr = \texttt{*@gmail.com} \wedge \texttt{'private'} \rightarrow deny \quad (7)$$

is associated with the variables v_{email}, $v_{\texttt{*@gmail.com}}$ and $v_{\texttt{private}}$.

One aspect of modelling that presents difficulty is how to handle outcomes $allow$ and $deny$. A single variable v_{allow} will not suffice, unless the set of policies happens to be such that exactly one policy matches each document (i.e. the policies happen to be normalised), giving a consistent instantiation of v_{allow} even if it appears in multiple constraints. The enforced action (i.e. $allow$ or $deny$) is modelled by means of a variable $v_{allow@i}$ which is $true$ if the policy with priority i allows the proposed action and $false$ if it disallows the proposed action. However if the policy at priority i has no opinion either way, $v_{allow@i}$ can be set either $true$ or $false$.

Each policy is converted into one or more clauses, depending on how complex it is. A simple policy like (7) converts straightforwardly into

$$\begin{aligned}&(v_{email} \wedge v_{\texttt{*@gmail.com}} \wedge v_{\texttt{private}} \rightarrow \neg v_{allow@2}) \quad (8)\\ \equiv &(\neg v_{email} \vee \neg v_{\texttt{*@gmail.com}} \vee \neg v_{\texttt{private}} \vee \neg v_{allow@2})\end{aligned}$$

assuming that it has priority 2. Hence when the left hand side of the policy is false, $v_{allow@2}$ can be either $true$ or $false$ but if the policy matches, it *must* be set to $false$ or else the clause has no literals in the solution.

However even a variable $v_{allow@i}$ for each priority i is not enough to model the problem, for supposing that $v_{allow@2}$ is $false$ in a solution: does this mean that policy (7) *forces* the action to be denied (excluding a higher priority rule that overrides it) or does it mean that clause (8) is already satisfied because the conditions don't match and $v_{allow@2}$ has been set arbitrarily?

To get around this a second type of variable is created, namely $v_{applies@i}$ for each priority level i. This variable is true if and only if the policy with priority i enforces an outcome. This must be modelled by adding clauses to the effect of

$$\text{LHS of policy} \leftrightarrow v_{applies@i}.$$

Now all that remains to do is to create a final variable v_{allow} and to create the following clauses:
If no rule applies then v_{allow} should be set to a default result of $true$ by default (corresponding to $allow$ by default):

$$\begin{aligned}&\bigwedge^{i} \neg v_{applies@i} \rightarrow v_{allow}\\ \equiv\ &v_{applies@1} \vee \ldots \vee v_{applies@maxprio} \vee v_{allow} \quad (9)\end{aligned}$$

If a policy at priority i applies and no higher priority policy applies, the final result is decided at i:

$$\forall i, v_{applies@i} \wedge \left(\bigwedge_{j=i+1}^{maxprio} \neg v_{applies@j}\right) \rightarrow v_{allow} = v_{allow@i}$$

which can be modelled clausally for an arbitrary i as

$$\begin{aligned}&\neg v_{applies@i}\\ \vee\ &v_{applies@i+1} \vee \ldots \vee v_{applies@maxprio}\\ \vee\ &\neg v_{allow@i} \vee v_{allow} \quad (10)\end{aligned}$$

$$\begin{aligned}&\neg v_{applies@i}\\ \vee\ &v_{applies@i+1} \vee \ldots \vee v_{applies@maxprio}\\ \vee\ &v_{allow@i} \vee \neg v_{allow}. \quad (11)\end{aligned}$$

These clauses correspond to the intuition that v_{allow} only has to be set to $true$ (resp. $false$) when the policy at i applies and forces the outcome to allowed (resp. disallowed) and every policy above i does not apply.

Example 5: We will now write down the SAT modelling of the following clauses from Example 3 and show their use.

$$email \wedge \texttt{'NewModel 5N'} \rightarrow deny \text{ (priority 1)}$$
$$email \wedge \texttt{'press release'} \rightarrow allow \text{ (priority 2).}$$

The variables needed are v_{email}, $v_{\texttt{NewModel\_5N}}$, $v_{\texttt{press\_release}}$, $v_{allow@1}$, $v_{allow@2}$, $v_{applies@1}$, $v_{applies@2}$ and v_{allow}. The clauses consist of

$$\begin{aligned}
\neg v_{email} \vee \neg v_{\texttt{NewModel_5N}} \vee \neg v_{allow@1} &\quad (12)\\
\neg v_{email} \vee \neg v_{\texttt{press_release}} \vee v_{allow@2} &\quad (13)\\
\neg v_{email} \vee \neg v_{\texttt{NewModel_5N}} \vee v_{applies@1} &\quad (14)\\
\neg v_{email} \vee \neg v_{\texttt{press_release}} \vee v_{applies@2} &\quad (15)\\
v_{email} \vee \neg v_{applies@1} &\quad (16)\\
v_{\texttt{NewModel_5N}} \vee \neg v_{applies@1} &\quad (17)\\
v_{email} \vee \neg v_{applies@2} &\quad (18)\\
v_{\texttt{press_release}} \vee \neg v_{applies@2} &\quad (19)\\
v_{applies@1} \vee v_{applies@2} \vee v_{allow} &\quad (20)\\
\neg v_{applies@1} \vee v_{applies@2} \vee \neg v_{allow@1} \vee v_{allow} &\quad (21)\\
\neg v_{applies@1} \vee v_{applies@2} \vee v_{allow@1} \vee \neg v_{allow} &\quad (22)\\
\neg v_{applies@2} \vee \neg v_{allow@2} \vee v_{allow} &\quad (23)\\
\neg v_{applies@2} \vee v_{allow@2} \vee \neg v_{allow} &\quad (24)
\end{aligned}$$

Clauses (12) and (13) model the policies. Clauses (14) to (19) ensure that variables $v_{applies@i}$ are set correctly. Clause (20) ensures that when no policy applies the outcome is $allow$. Clauses (21) and (22) ensure that when policy 1 applies and 2 does not then the overall outcome is the required protection

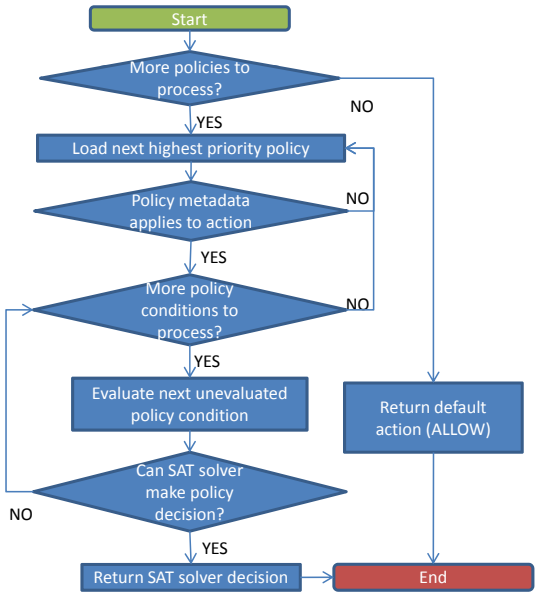


Fig. 3. Flowchart of constraints-based decision procedure

of policy 1. Clauses (23) and (24) ensure that when policy 2 applies then the overall outcome is the required protection of policy 2.

Suppose that the user attempts to email and their document contains the word "NewModel 5N" but not "press release". We will choose to set variables v_{email} and $v_{\texttt{NewModel\_5N}}$ to $true$ and $v_{\texttt{press\_release}}$ to $false$ to reflect this information about the user action. v_{allow} is set to $true$ so that if the action is allowed a solution will be found, but if it's not allowed it is impossible to complete the assignment and the solver will report that there is no solution. Now we will ask the SAT solver to find a solution. Because of clauses (12) and (14) variable $v_{allow@1}$ must be set to $false$ and $v_{applies@1}$ must be set to $true$. By clause (19) $v_{applies@2}$ must be set to $false$. Hence by clause (22) v_{allow} has to be $false$. However v_{allow} cannot be both $false$ and $true$ and so no solution is possible, and the solver must succeed in proving that the action should be stopped (we do not specify how it achieves this). ■

B. Policy Decisions on User Actions

Example 5 described how policy decisions can be made using a constraint model of the policied. In practice, however, it would be preferable to avoid loading all the policies into the SAT solver and also preferable to be able to avoid evaluating all the conditions. This is especially true if the condition is hard to check, e.g. the presence of a string in a very long document.

In this section, we will describe how policy decisions are implemented and demonstrate that our implementation is fast enough for practical use by testing it on a wide range of randomly generated sets of policies. Our criterion is that the delay of checking policy conditions should be barely perceptible and we set ourselves a target of 0.25 seconds for making decisions.

Our implementation works as shown in Figure 3. In short, policies are loaded one by one, and conditions are evaluated until the SAT solver is able to prove definitively what protection must be enforced. In this way, only the policies and conditions that are necessary to get a result are processed. In the current implementation, policies are preloaded into memory before the procedure begins, but this may vary on a different architecture with less memory.

1) Empirical evaluation: We have carried out an empirical evaluation of the efficiency of our SAT implementation. We have obtained 20 text documents from `gutenberg.org` ranging in size from 1Kb to 5Mb. From these we have generated random databases of policies. The condition on each policy is a randomly chosen disjunction of up to a maximum mp disjuncts. Each condition is picked randomly from either an English dictionary or the text document from which the database is derived. The path under which the policy applies is also chosen at random. The protection is also chosen at random from a choice of $allow$ or $block$ (without embellishment). The final parameter to our random generator is the number of policies in the database. Overall we have 540 such databases under test.

In order to test the database, we attempt to save the document the database was generated from to a randomly chosen location on disk. Then the SAT decision engine is asked to decide whether this attempt is allowed or not.

Table I summarises our results. We have split the results up into rows for each database size and maximum condition length. The "Decision time" columns are summary statistics for the total time taken to make a decision for a single document, including the time to load the file from disk. They demonstrate that we have been successful in keeping decision time to a level which does not inconvenience users. The "Proportion of decision time spent searching" columns give the proportion of the decision time that is spent evaluating conditions, i.e. searching for relevant words in the document text. The median is consistently around a third and the maximum around $\frac{4}{5}$. This suggests that in our implementation, there is no major overhead either in evaluating conditions or in running the SAT solver. Finally in the "Number of strings searched" columns we show that very few strings need to be searched for in order to make a decision. This is because we load and evaluate only the minimum number of policies needed to form a decision. If, for example, every condition in a 1000 policy database is checked, that is a minimum of 1000 conditions and possibly many more, but we do it in a maximum of 6 checks for any action.

VII. Description of Intelligent Policy Assistant

In this section we will describe our assisted policy modelling environment which we use to help the user to define policies such that

- redundant policies are avoided (i.e. policies that can be removed without changing the protection for any action);
- they understand the effect of adding, editing and deleting a policy; and

TABLE I
TABLE OF STATISTICS ON POLICY DECISIONS

Policy set		Decision time			Proportion of decision time spent searching			Number of strings searched		
Size	Cond. size	Min.	Med.	Max.	Min.	Med.	Max.	Min.	Med.	Max.
10	5	0.000	0.018	0.178	0.000	0.361	0.729	0	1	5
10	10	0.000	0.020	0.661	0.000	0.399	0.811	0	1	7
10	15	0.000	0.018	0.269	0.000	0.364	0.764	0	1	4
100	5	0.003	0.024	0.507	0.045	0.469	0.834	1	1	6
100	10	0.004	0.022	0.243	0.024	0.436	0.779	1	1	5
100	15	0.003	0.023	0.299	0.060	0.396	0.759	1	1	5
1000	5	0.002	0.025	0.331	0.053	0.362	0.700	1	1	3
1000	10	0.002	0.025	0.254	0.008	0.346	0.812	1	1	6
1000	15	0.003	0.022	0.265	0.020	0.385	0.717	1	1	4

- they are assisted in choosing priority so that they get their desired behaviour during policy contradictions.

We wish to avoid redundant policies because they slow down policy decisions (the engine has to maintain policies that make no difference) and because we would prefer that users could avoid expending mental effort to understand redundant policies. Users need to be aided in understanding their policies because if they make an unintended change to their policies they may introduce a potential data leak or make certain permissible business processes impossible to complete.

A. Supported workflows

We support operations to add, edit and remove policies from the database. For usability reasons, we restrict the user to one operation at any one time. This is because any two policies with incompatible protections can interact to create a contradiction (as shown in Figure 1) and this includes any policies the user is currently editing. Hence any change they make in one policy may effect the meaning of the other. To avoid this unpredictable situation we allow one policy to be processed at a time, so that there is an unchanging background set of policies and no chance of unpredictable side-effects.

A key part of the modelling assistant is summarising the effect of policies both individually and in groups. We will describe how we achieve this in the next section.

B. Summarising policies

Suppose we have one of the following problems:

- For a particular policy p, we want to know *pertinent* and *exhaustive* examples of actions, metadata and documents along with the protection p enforces on those documents. By pertinent we mean using only key words that appear in the policy and by exhaustive we mean that we want to know about all classes of documents the policy applies to but definitely *not* every document because they are infinite in number.
- For a pair of policies p and q, we want to know, by pertinent and exhaustive examples, what happens if p is higher priority than q and vice versa.
- For a policy p and an edited version p' of the same policy, we want to know pertinent and exhaustive examples of the actions whose protection *differs* between before and after p is edited.

Our next example demonstrates 'pertinent' and 'exhausive':

Example 6: Consider the policy

$$email \wedge (\texttt{'private'} \vee \texttt{'confidential'}) \rightarrow allow.$$

The pertinent words are "private" and "confidential". Since the condition is a disjunction, the policy applies if either "private" or "confidential" or both is in the document, and when it applies the outcome is $allow$. This is 3 pertinent classes of document, and the corresponding examples are:

Document contains	Protection
`'private'`	$allow$
`'confidential'`	$allow$
`'private','confidential'`	$allow$

The first line means that any document containing 'private' is allowed, for example "private parking" and "private property". When a pertinent word is not included in an example, it implicitly should not be included in the document. ■

The previous example showed how we summarise the effects of a single policy. However we also wish to be able to summarise what happens when two policies apply in combination, or when a policy is changed. For the former, we find pertinent examples of documents to which one or both policies apply, and in case of contradictions the protection required is decided by the higher priority.

Example 7: Suppose we want to summarise the effect of the following policies: the priority 1 policy

$$email \wedge \texttt{'NewModel'} \wedge \texttt{'5N'} \rightarrow deny, \text{ and}$$

$$email \wedge (\texttt{'declassified'} \vee \texttt{'press release'}) \rightarrow allow$$

with priority 2. The pertinent words are "NewModel", "5N", "declassified" and "press release". The relevant examples are

Document contains	Protection
`'NewModel','5N'`	$deny$
`'NewModel','5N','declassified'`	$allow$
`'NewModel','5N','press release'`	$allow$
`'NewModel','5N', 'declassified','press release'`	$allow$
`'declassified'`	$allow$
`'press release'`	$allow$
`'declassified','press release'`	$allow$

There is one way for the first policy alone to apply. There are 3 ways for the second policy alone to apply. There are a further 3 ways for both to apply at once. Hence 7 examples are given. ■

We will now describe how we generate examples. We described above in §VI that we implement policies using constraints. To enumerate outcomes for a couple of policies we

simply post them both in the way described in that section, and then ask the SAT solver the generate all solutions such that a policy applies, i.e. we add the clause $v_{applies@1} \vee v_{applies@2}$. To generate an example for each solution, we (1) find all variables representing policy conditions that have been set to $true$ in the solution, these are the terms the document must contain, (2) find metadata whose corresponding variables have been set to $true$ to see, for example, which save path or which email recipients the example pertains to and (3) check the value of v_{allow} to find out if that document is allowed ($true$) or denied ($false$) by that pair of policies. This implementation has the advantage that it is entirely decoupled from the meaning of policies, provided that the policy has been modelled as SAT clauses, this enumeration routine simply requests all solutions and interprets them. In an imperative approach it would require some care to ensure that all examples were found and that the semantics of policies were taken into account even in the presence of complex conditions including arbitrary boolean operations.

Next we describe how priorities are depicted in the policy editor interface.

C. Maintaining priorities

In the policy editor interface, we do not directly expose the priority relations between policies as shown in Figure 2. Instead we display the policies in a table, ordered from highest to lowest priority. We believe that this makes it easy for users to see at a glance what policies are high and low priority. The ordering in the table is equivalent to a preorder traversal of the priority graph, meaning that whenever a there is a path in the graph from policy p to policy q, p must be earlier in the list than q. However, in the next sections we describe the policy assistant where we take into account the specifics of the underlying priority graph, and make automatic ordering decisions on the user's behalf wherever possible based on it.

We will now describe with examples how our modelling assistant helps the user to achieve the add, edit and remove workflows.

D. Add policy

The add policy function is implemented as a wizard, presenting the user with a series of choices until the system has determined what the policy is, how it should interact with other policies and whether the policy database can be simplified by removing newly redundant policies as a result. However changes to the database are only committed at the end of the wizard, so it can be used for exploratory modelling.

The wizard is structured as follows (1) define the new policy, i.e. the conditions, details of matching actions and the required protection, (2) notionally add the new policy at top priority, (3) ask the user to make a series of choices, in order to move the policy down in priority until it reaches a position that they are happy with, (4) summarise the effect of the change and finally (5) commit the change to the policy database.

Steps (1), (2) and (5) seem relatively self-explanatory, but we will give algorithms and justification about how we achieve (3) and (4).

In step (3) at each step we have a new policy called p and a current policy called cp, which is the next policy below p in the list described in §VII-C. Each step, the wizard analyses policies p and cp in order to decide what needs to happen:

leave p at bottom priority and stop if p has already reached the bottom and hence there are no more policies left to process, i.e. cp is null/undefined

discard p and stop if protections of p and cp are the same, but cp has a more general condition, i.e. applies to every action that p applies to. In other words p is redundant.

delete cp and continue if protections of p and cp are the same, but p has a more general condition. In other words cp is redundant, and it might as well be replaced by p. However we continue to make more decisions about moving p down the priority, because p covers a broader condition and its additioal interaction with other policies should be checked by the user.

skip over cp if p and cp have compatible protections or never apply to the same action. In other words it doesn't matter what order they come in so we do not bother asking the user to decide.

pick either p or cp and continue if the protections of p and cp are incompatible and the conditions match exactly the same documents. The user is asked to pick one or the other, because a contradiction must result when they apply. This case is almost certainly the result of user error, so they might also choose to cancel the wizard or go back and change the policy.

automatically leave p in current position and stop if p and cp are incompatible but the condition of cp is more general. This is done because if p is placed below cp then the result of p will always be ignored so putting it above is the only sensible option. Putting p below cp is called *shadowing* in [6] and should be avoided.

user picks whether p or cp should have higher priority in all other cases the user is shown the difference between putting p above cp or cp above p. They have a free choice to do either depending on which of the examples the system presents are correct.

The system either makes an automatic choice or examples are displayed to illustrate the actions available to the user (as described in §VII-B).

Once the process stops, the user has chosen a position for their new policy in the ordering. The system need not ask the user to make decision about the ordering of p against any policy lower than cp, because the result is implied.

For step (4) in the wizard we aim to present a final summary to the user, as a last step before they commit their changes. Recall that in step (3) the policy is placed into the ordering at some position pos. The summary we present is to give pertinent and exhaustive examples of what happens when p and each policy lower than p apply together.

Example 8: Suppose $save \wedge \texttt{technical} \wedge \texttt{report} \rightarrow allow$ (1) and $save \wedge \texttt{NewModel} \wedge \texttt{5N} \rightarrow deny$ (2) are already posted into the policy database. These policies together mean that technical reports are always allowed but that documents

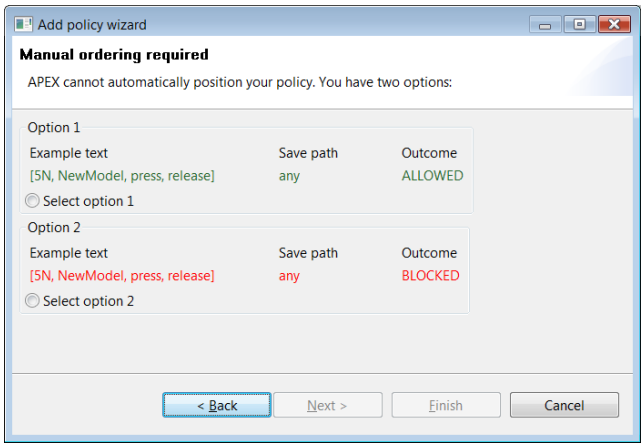


Fig. 4. Assisted policy ordering screen

containing the name of a new product "NewModel 5N" cannot be saved. Suppose the user wishes to add a special case that press releases should also be allowed. Hence the new policy p=$save \wedge \texttt{press} \wedge \texttt{release} \rightarrow allow$ is added. Policy (1) is skipped as it has the same protection as the new policy and their relative is order is unimportant. However the user is asked to manually order p and policy (2). The screen is shown in Figure 4. The user picks the example that corresponds to what they want to happen in practice, i.e. they pick the example where saving documents containing "NewModel", "5P", "press" and "release" is allowed rather than denied. ■

E. Remove policy

Now we will describe the remove workflow. When removing a policy p the remaining policies can be split into three classes:

- Those that have a protection compatible with p's protection, these policies are unaffected by p's removal because if they both apply the outcome is unchanged.
- Those that are higher priority than it, these policies are unaffected by the removal of p, because they already override it.
- The remaining policies, i.e. those with a protection incompatible with p and a lower priority. The effect of such a policy, let us call it q, are potentially changed on documents that both p and q apply to.

Our aim is to help the user understand what actions have their security protection changed by removing p. In the wizard this is achieved by displaying pertinent and exhaustive examples of actions whose protections differ before and after p is removed, for each policy in the final group above, i.e. those whose protections are incompatible with and have a lower priority than p.

Example 9: In descending order of priority we have policies $save \wedge \texttt{technical} \wedge \texttt{report} \rightarrow allow$, $save \wedge \texttt{press} \wedge \texttt{release} \rightarrow allow$ and $save \wedge \texttt{NewModel} \wedge \texttt{5N} \rightarrow deny$. Suppose we wish to delete the first policy. This doesn't affect the second policy, because they both have the same protection. However the third policy loses its effective coverage of documents containing both "technical report" and "NewModel 5N",

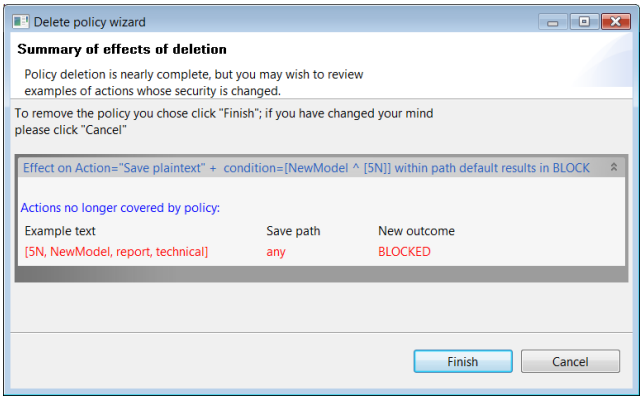


Fig. 5. Assisted policy deletion screen

meaning that saving such documents goes from being allowed to being denied. The wizard summary is shown in Figure 5. ■

F. Edit policy

Editing a policy is more or less decomposed into a deletion followed by adding a fresh policy. We have made the wizard comparatively similar to the add policy wizard, so that when the user becomes comfortable with adding policies they should be comfortable with editing too. The disadvantage of this is that the user may have already made decisions about what should happen in the case of conflicts, and this knowledge is not carried over to apply to the edited policy.

The steps in the edit policy wizard are (1) edit the policy particulars (2) summarise the change in the policy, i.e. give examples of what it did before and what it does after (3) ask the user to make a series of choices, in order to move the policy down in priority until it reaches a position that they are happy with, (4) summarise the effect of the change and finally (5) commit the change to the policy database.

Apart from step (2), these are the same as for adding a policy. In step (2) we display a set of pertinent and exhaustive examples of the policy before the edit, and after the edit. The user can therefore see the isolated effect of the policy changes, i.e. neglecting policy contradictions and looking only at how it applies on its own.

Example 10: Suppose that the user notices that policy $save \wedge \texttt{techical} \wedge \texttt{report} \rightarrow allow$ has a spelling mistake in it, and corrects it. The initial summary displayed is as shown in Figure 6. ■

G. Implementing the Intelligent Policy Assistant

In §VII-B we described how a SAT solver is used to generate policy examples. Now we describe how it is used to implement some other features of the add, remove and edit policy wizards. In all these procedures there is frequently a need to discover when a policy p has a more general, more specific or equal condtion compared to another policy q.

With a SAT solver it is very easily to implement this functionality. To show that p is at least as general as q, we will ask the SAT solver to find a counterexample, i.e. can it find a document such that q applies but p does not. This involves

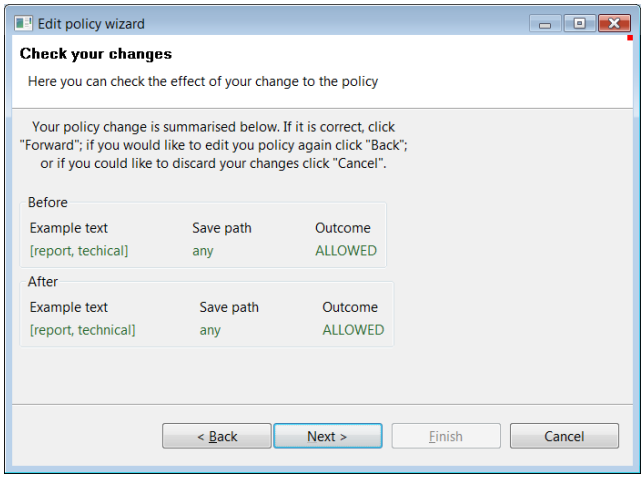


Fig. 6. Assisted policy editing screen

posting both p and q as clauses so that variables $v_{p-applies}$ and $v_{q-applies}$ represent whether p and q apply respectively. Now set $v_{p-applies}$ to $false$ and $v_{q-applies}$ to $true$. If the solver cannot find a solution is has proved that p is at least as general as q.

To prove that p is at least as specific as q, simply prove that q is at least as general as p as described above. To prove that p has the same condition as q, prove that p is at least as general as q and q is at least as general as p.

VIII. Conclusions

We have succeeded in breaking new ground on several fronts. First, we have shown that a constraint programming implementation of policy decisions has fast performance when making policy decisions. Second, we have shown that a constraints representation allows us to implement a rich modelling environment for helping the user to create policies. The SAT solver provides us with facilities to enumerate documents based on policies and to prove features of policies, both of which are needed to create a functional modelling environment for policies. Finally, we have shown that a policy modelling environment can make it substantially easier for a user to create and understand security policies, meaning that they are aware of side effects and interactions between their policies. They are supported in making ordering decisions on their policies, with the drudgery removed and guesswork being replaced by informed decisions.

References

[1] H. Balinsky, D. S. Perez, and S. Simske, "System call interception framework for data leak prevention," in *Proceedings of IEEE International EDOC Conference*, 2011, in press.

[2] S. J. Simske and H. Balinsky, "APEX: Automated policy enforcement eXchange," in *ACM Document Engineering*, 2010, pp. 139–142.

[3] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo, "A logical framework for reasoning on data access control policies," *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pp. 175–189.

[4] E. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852–869, 1999.

[5] M. Ryan, "Defaults in Specifications," in *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. IEEE, pp. 142–149.

[6] E. Al-Shaer and H. Hamed, "Firewall Policy Advisor for anomaly discovery and rule editing," *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003.*, pp. 17–30.

[7] S. Benferhat, R. El Baida, and F. Cuppens, "A stratification-based approach for handling conflicts in access control," *ACM Symposium on Access Control Models and Technologies*, p. 189, 2003.

[8] H. Hamed and E. Al-Shaer, "Taxonomy of conflicts in network security policies," *IEEE Communications Magazine*, vol. 44, no. 3, pp. 134–141, Mar. 2006.

[9] A. Heydon, M. Maimone, J. Tygar, J. Wing, and A. Zaremski, "Miro: Visual specification of security," *Software Engineering, IEEE Transactions on*, vol. 16, no. 10, pp. 1185–1197, 1990.

[10] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino, "A unified framework for enforcing multiple access control policies," in *ACM SIGMOD Record*, vol. 26, no. 2. ACM, 1997, pp. 474–485.

[11] G. Russello, C. Dong, and N. Dulay, "Authorisation and Conflict Resolution for Hierarchical Domains," *Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*, pp. 201–210, Jun. 2007.

[12] C.-c. Shu, E. Y. Yang, and A. E. Arenas, "Detecting Conflicts in ABAC Policies with Rule-Reduction and Binary-Search Techniques," *2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, pp. 182–185, Jul. 2009.

[13] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible support for multiple access control policies," *ACM Transactions on Database Systems*, vol. 26, no. 2, pp. 214–260, Jun. 2001.

[14] N. Li and J. Feigenbaum, "IBM Research Report A Logic-based Knowledge Representation for Authorization with Delegation," *New York*, vol. 21492, no. May, 1999.

[15] H. Kamoda and M. Sloman, "Policy Conflict Analysis Using Free Variable Tableaux for Access Control in Web Services Environments," *Policy*, 2005.

[16] S. Davy, B. Jennings, and J. Strassner, "Efficient Policy Conflict Analysis for Autonomic Network Management," *Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (ease 2008)*, pp. 16–24, Mar. 2008.

[17] ——, "Application domain independent policy conflict analysis using information models," *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, pp. 17–24, 2008.

[18] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict resolution using logic programming," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 1, pp. 245–250, Jan. 2003.

[19] D. Agrawal, J. Giles, and J. Lobo, "Policy Ratification," *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pp. 223–232.

[20] A. Schaad and J. D. Moffett, "The Incorporation of Control Principles into Access Control Policies," *System*, no. 99311141.

[21] J. Hwang, T. Xie, and V. Hu, "ACPT : A Tool for Modeling and Verifying Access Control Policies," *Policy*, pp. 2–5.

[22] S. Bistarelli and S. Foley, "Analysis of integrity policies using soft constraints," *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 77–80.

[23] L. Ramshaw, a. Sahai, J. Saxe, and S. Singhal, "Cauldron: A Policy-Based Design Tool," *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pp. 113–122.

[24] J. Chomicki, J. Lobo, and S. Naqvi, "A Logic Programming Approach to Con ict Resolution in Policy Management," *Syntax*.

[25] C. Ribeiro, P. Ferreira, C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, "Security Policy Consistency," *Security*.

[26] a.K. Bandara, E. Lupu, and a. Russo, "Using event calculus to formalise policy specification and analysis," *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 26–39.

[27] A. Jeffrey and T. Samak, "Model Checking Firewall Policy Configurations," *2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, no. 1, pp. 60–67, Jul. 2009.

[28] "eXtensible Access Control Markup Language (XACML) Version 3.0," OASIS Access Control TC, Tech. Rep., 2010. [Online]. Available: http://tinyurl.com/5vxksvk

[29] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, ser. STOC '71, 1971, pp. 151–158.

[30] D. Le Berre and A. Parrain, "The SAT4J library, release 2.2," *JSAT*, vol. 7, pp. 59–64, 2010.