Policy Ratification

Dakshi Agrawal, James Giles, Kang-Won Lee, Jorge Lobo
IBM T. J. Watson Research Center
{*agrawal,gilesjam,kangwon,jlobo*}*@us.ibm.com*

Abstract

It is not sufficient to merely check the syntax of new policies before they are deployed in a system; policies need to be analyzed for their interactions with each other and with their local environment. That is, policies need to go through a ratification *process. We believe policy ratification becomes an essential part of system management as the number of policies in the system increases and as the system administration becomes more decentralized.*

In this paper, we focus on the basic tasks involved in policy ratification. To a large degree, these basic tasks can be performed independent of policy model and language and require little domain-specific knowledge. We present algorithms from constraint, linear, and logic programming disciplines to help perform ratification tasks. We provide an algorithm to efficiently assign priorities to the policies based on relative policy preferences indicated by policy administrators. Finally, with an example, we show how these algorithms have been integrated with our policy system to provide feedback to a policy administrator regarding potential interactions of policies with each other and with their deployment environment.

1 Introduction

In recent years, we are witnessing a growing interest in using policies for system management. In the literature, policy-based management has been proposed for a variety of IT tasks such as access control, data backup, network security, resource provisioning, configuration checking, and service planning [1, 8, 4, 2, 20, 11]. The goal of a policy-based management system is to allow an IT administrator to define high level directives to enforce business rules and objectives, instead of writing customized scripts, or manually configuring and auditing individual resources.

It has been long recognized that merely providing a policy editing tool to ensure correct policy syntax is not sufficient. Policies can interact with each other, often with undesirable effects, and an IT administrator needs to be aware of such interactions among policies. The problem of policy interaction is particularly acute in a distributed system where it is likely that a policy author would have only a partial view of the entire system and where multiple authors may write policies applicable to the same set of resources [2].

We say a policy P is *ineffective* in a domain if there is no resource in the domain whose operation will be affected by this new policy or if the policy P is *dominated* or *shadowed* by the already existing policies applicable to the domain. For example, if a local password policy already mandates that "the password length must be greater than 8", then an incoming global policy "the password length must be greater than 6" will not change the way the system is being managed locally. Ideally any ineffective policy must not be deployed or marked inactive so that CPU cycles are not wasted on evaluating it.

We say that two policies are *in conflict*, if, under certain circumstances, they may issue directives that cannot be achieved simultaneously. For example, if one policy says that "all Windows workstation must check and install the latest service pack from Microsoft every week" and another policy says that "the Windows XP Service Pack 2 must not be installed (due to its incompatibility with existing firewall software)", then, at the time when the Windows XP Service Pack 2 has been released, the two policies conflict with each other because their goals cannot be achieved simultaneously. In certain cases, meta-level rules can be used to resolve conflicts in an automated manner [17, 22, 15, 14]. However, in many cases, it is necessary for the human administrators to resolve conflict since it is not always clear which policy should win among the conflicting ones. In general, conflicts can be resolved either by specifying priorities among conflicting policies or by disabling one of the conflicting policies. In the above example, the human administrator may specify that the latter policy has high priority so that in general Windows workstations are up to date, but the installation of Service Pack 2 is prohibited.

In this paper, we study the problem of *policy ratification*—the process by which a new policy is approved before being committed in a system by taking into account its potential interactions with other policies and its deployment environment. Our focus in this paper is on discipline independent ratification tools that can be a part of generic policy middleware. The study of interaction among policies that requires discipline-specific information is outside the scope of this paper. Specifically, the key contributions of this paper are the following:

- We identify the primitive operations that can be used for policy ratification regardless of the discipline, namely *dominance check*, *potential conflict check*, *coverage check*, and *consistent priority assignment*.

- We present the algorithms to implement these primitive operations. In particular, we identify five classes of boolean expressions for which we provide effective algorithms to implement the operations listed above. We also present an efficient algorithm to assign priority to a policy that is consistent with the relative preferences of policies specified by a system administrator.

- Finally, we describe how policy ratification can be used in real life scenarios with print service policies as an example. To help understand how policy ratification can be used, we present a screen shot of the graphical user interface designed on top of our policy ratification module.

The remainder of this paper is organized as follows: Section 2 briefly presents the policy models considered in this paper, and the primitive operations required by a generic policy ratification module. Section 3 presents algorithms to implement ratification operations. Section 4 gives an example of how policy ratification can be used for defining print service policies. Section 5 briefly reviews the related research in this area, and compares it with our work. Finally, Section 6 contains the main conclusions of this paper, and the future research directions.

2 Preliminaries

Policies[1] can be defined in terms of a *guideline* or a *goal* by giving a statement that can be evaluated to be either true or false. For instance, the policy "the average response time of a web server measured over an hour period should be less than 15 msec", is an example of such a policy. In this paper, we call these "goal" policies.

Another type of policy is related to system configuration. In this case, policies are generally specified as key-value pairs. One example of such policies is the configuration policies for the Microsoft Exchange server. For example, the value of `DisableCollaborationApps`, associated with the key `HKEY_CURRENT_USER\Software\Policies\Microsoft\Messenger\Client`, determines whether or not the whiteboard and application sharing features of Windows Messenger are enabled. We call such policies "configuration" policies.

There are more sophisticated policy languages to handle complex systems management tasks [8, 16, 19, 1]. Ponder, described in [8], specifies policies in a *subject-action-target* (or SAT) format.[2] The *mode* of a policy specifies either positive **O+** (must do) or negative **O-** (must not do) obligations, or positive **A+** (allowed to do) or negative **A-** (not allowed to do) authorizations. The *subject* specifies the human or automated managers and agents to which the policy applies and specifies the entities which interpret obligation policies. The *target* specifies the objects on which actions are to be performed. The *actions* specify what must be performed for obligations and what is permitted for authorizations. An optional field called *constraint* specifies a boolean condition when the policy is applicable. For example, the policy: "all corporate users (the subjects) are allowed (the mode) to browse and purchase (the action) a shared travel-booking agent (the target)" can be easily expressed in Ponder (see [17] for more details).

There are other policy models that follow a general format of *event-condition-action* (ECA) [16, 19, 1]: "when *event* occurs, if *condition* is true then execute *action*." The policy core information model (PCIM)[19] by IETF is an instance of the ECA policy model. ACPL (autonomic computing policy language) [1] and PDL (policy definition language) [16] are also based on the ECA policy model.

While there are significant differences in the various types of policies (goal, configuration, SAT, and ECA types), a few characteristics are common to all. First, the evaluation of a boolean expression is a key operation required to evaluate these policies. For example, goal policies are boolean expressions. Configuration policies can be thought of as a conjunction of equalities. The condition part of ECA policies and the constraint part of SAT policies are boolean expressions. Second, these policies contain variables and may specify actions. Variables and actions that are eligible to appear in a policy are determined by the *scope* of the policy. Typically, there are multiple policies defined for a scope. For some assignment of values to the variables, multiple policies may be applicable and they may issue conflicting directives.

2.1 Policy ratification

Regardless of the policy model used in a policy system, when a new policy is written or committed in the system, the administrator must consider how the new policy interacts with those already existing in the system. In addition, given a group of policies, the administrator may want to know if it provides sufficient guidance for a system. We define these tasks performed by system administrators as *policy ratification*[2]. Based on the result of policy ratification, administrators would accept or reject new policies, assign priorities to resolve potential conflicts, or mark certain policies as being inactive. In particular, we have identified the following generic operations that may be performed during policy ratification:

Dominance Check A policy x is dominated by a group of policies $Y = \{y_1, \ldots, y_n\}$ ($n \geqslant 1$) when the addition of x does not effect the behavior of the system governed by Y. For example, a policy "password length $\geqslant$ 6" is dominated by another policy "password length $\geqslant$ 8" because the former policy is subsumed by the later. In another example, a pol-

[1]Defining and classifying different policy models is a challenging research topic in itself. In this paper, we review some existing policy models to motivate operations required for policy ratification without trying to come up with a comprehensive taxonomy.

[2]In addition, a Ponder policy can have optional fields such as constraint, trigger, exception, etc.

icy "Joe has access to machine X from 1 P.M. to 5 P.M." is dominated by another policy "Joe has access to machine X from 8 A.M. to 7 P.M." From these examples, we observe that determining whether a boolean expression implies another boolean expression is a crucial ratification operation: in the first example, we need to determine that $(p.length \geqslant 8) \Rightarrow (p.length \geqslant 6)$, while for the second example, we need to determine that $(1300 \leqslant t \leqslant 1700) \Rightarrow (0800 \leqslant t \leqslant 1900)$.[3]

Conflict Check We say two goal policies are in conflict when they specify goals (as boolean expressions) that cannot be satisfied simultaneously (e.g. "password length > 8" and "4 $\leqslant$ password length $\leqslant 8$"). For configuration policies, we say two policies conflict when they specify different configuration values: "disk-quota=2 GB" and "disk-quota=1 GB". If the configuration parameters can take a range of values, then conflict among configuration policies can be defined in a similar manner as for the goal policies. In the ECA model, a *potential* conflict between two policies may arise when the conditions of two policies can be simultaneously true, i.e., both policies may become applicable, and *may* specify two incompatible actions. Therefore, the key ratification operation here is to determine whether a conjunction of two boolean expressions is satisfiable.

In this paper, we assume that potential conflicts among policies would be resolved by human administrators either by marking policies inactive or by indicating relative priorities of the conflicting policies.

Coverage Check In many application domains, the administrator may want to know if explicit policies have been defined for a certain range of input parameters. For example, in the ECA model, the administrator may want to make sure that regardless of the values of input parameters, at least one policy has a true condition (see Section 4 for a concrete example). The key operation in this case is to find out if a disjunction of boolean expressions implies another boolean expression.

Consistent priority assignment Most policy systems prioritize policies by assigning an integer value to each policy. Thus in case several policies apply, the one with the highest priority is executed. The manual assignment of priorities to the policies can work when the number of policies is small. However, if the number of policies is large, the manual priority assignment becomes infeasible. Moreover, if the assignment of priorities is not done with care, then the insertion of a new policy may require reassignment of priorities to a large number of policies.

Figure 1 depicts one such example. Let each vertex in the graph denote a policy with an integer priority, and each arrow in the graph connect two potentially conflicting policies with the arrow going from the policy of higher priority to the policy of lower priority. When a new policy is added (denoted by the white vertex), its relative priority is specified with respect to the other policies in the system as shown in the figure. In this particular example, the new policy cannot be assigned a priority without reassigning priorities to policies already existing in the system.

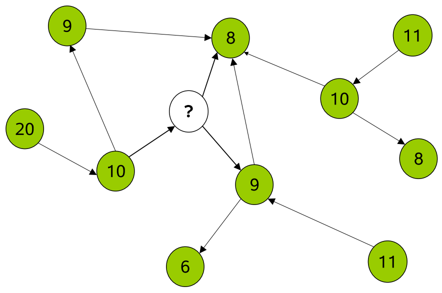


Figure 1. Relative Priority Graph of Policies

The reassignment of priorities to the existing policies needs to be done carefully to avoid a large (average as well as peak) overhead during insertions; especially if the updated policies need to be disseminated in a distributed system. The goal of consistent priority assignment is to take *relative* preferences (or in other words, the priority graph) specified by the policy author, including those specified during dominance and conflict checks, and assign integer priorities to the policies so that the number of amortized reassigned priorities is minimized.

In summary, we have identified four critical high-level ratification operations important for a wide range of policy models. Although the details of how these high-level operations are exposed to the user would depend on the policy model and its application context, the low-level operations involved are largely independent of the policy model and its application context. In particular, we need algorithms to find satisfiability of boolean expressions (and if possible the set of all values for which boolean expression evaluates to true), and an algorithm to assign priorities to policies so that amortized number of priority reassignments can be minimized. In the next section, we will discuss these algorithms.

3 Algorithms

To facilitate discussion, we will start by defining a few terms and notation. A boolean expression is *atomic* if it does not contain explicitly compound logical operators $(\wedge, \vee, \oplus$ etc.); a boolean expression is *compound* otherwise. For a given regular expression r, $L(r)$ denotes the language generated by the regular expression r [13]. Thus expressions $(\text{apple} \neq \text{orange}), (5 \leqslant x \leqslant 10)$, and $(\texttt{username} \in L(\text{"joe*"})$, are all atomic boolean expressions, while the expression $(5 \leqslant x) \wedge (x \leqslant 10)$ is a compound boolean expression.

There are many challenges in determining whether a boolean expression can be `true` for an assignment of values to the variables. This is an extension of the classic *NP*-complete satisfiability problem where the atomic expressions

[3]Note that the interpretation of the dominance depends on the semantics of the policy as in the first example the implied boolean expression is dominated while the case is opposite for the second example. However, in each case, logical implication is the fundamental operation.

are restricted to boolean variables [13]. There are few known classes of formulas that are tractable. Horn clauses is one, where the disjunctive normal form of the boolean expression has at most one positive variable. It is tractable too if the number of variables per disjunct in the disjunctive normal form is limited to two. However, more than two variables makes the problem *NP*-complete. Even worse, we are sometimes interested in showing that two formulas are not simultaneously satisfiable, which is a *co-NP* problem.

Our examination of policies from many different application domains such as network QoS, storage, database, and identity management shows that often policies involve only the subclasses of boolean expressions for which the satisfiability problem is tractable. Thus, our approach is to identify categories of boolean expressions that occur frequently in typical policies and address the satisfiability problem for such cases. In the rest of this section, we describe five common categories of boolean expressions and algorithms to solve the corresponding satisfiability problem.

Category 1: Real valued linear constraints. The general form of boolean expressions in this category is $\bigwedge_{i=1}^{n} B_i$, $B_i = \sum_{j=1}^{m} a_{ij}x_j \rhd c_i$, where $a_{ij} \in \mathbb{R}$ are constants, $x_j \in \mathbb{R}$ are variables, and $\rhd \in \{=, <, \leqslant, >, \geqslant\}$. In other words, this category contains conjunctions of atomic boolean expressions defined by *linear constraints* in m-dimensional real space. We use a modified simplex algorithm to find satisfiability for these expressions.

Category 2: One variable inequality constraints. The general form of boolean expressions in this category is $\bigwedge_{i=1}^{n} B_i$, $B_i = x \rhd c_i$, where c_i is a constant, x is the variable, and $\rhd \in \{=, <, \leqslant, >, \geqslant\}$. The variable x and constants $c_1, \ldots, c_n$ are of data type `real`, `integer`, `string`, or `calendar`[4]. Note that for `real` and `integer` data types linear constraints of the form $a'_i \cdot x + b'_i \rhd c'_i$ can be reduced to the form $x \rhd c_i$. For expressions in this category, we use the domain elimination algorithm to find satisfiability as well as the value of variables for which the boolean expression is `true`.

Category 3: One variable equality constraints. The form of boolean expressions in this category is $\bigwedge_{i=1}^{n} B_i$, $B_i = x \rhd c_i$, where c_i is a constant, x is the variable, and $\rhd \in \{=, \neq\}$. The variable x and constants $c_1, \ldots, c_n$ are either of data type `composite`[5] or of data type `boolean`. For this category also, we can use the domain elimination algorithm to find satisfiability as well as the value of variables for which the boolean expression is `true`.

Category 4: Regular expression constraint. The general form of boolean expression in this category is any boolean expression formed using $\wedge$ and $\vee$ with atomic expressions of the form either $s \in L(r)$ or $s \notin L(r)$, where s is the same string variable in all the expressions, r is a regular expression, and $L(r)$ is the language generated by regular expression r. For expressions in this category, we can use finite automata techniques to find satisfiability [13].

Category 5: Compound boolean expression constraint. This is the most general category that includes expressions obtained by compounding boolean expressions belonging to the categories listed above. Without loss of generality, we can assume that the logical operators used for compounding are $\wedge, \vee$, and $\neg$. For expressions in this category, we use a solution tree algorithm that iteratively produces conjunctions that constitute disjunctive normal form of the given boolean expression. This class is the only intractable class we present and we address it with heuristics.

We are not going to address in this paper in how to deal with the Regular expression constraint category. We just mention that $\vee$ connectors can be treated as unions and $\wedge$ connectors as intersections of regular expressions so that the boolean expression that we might extract from the policies can be translated into the question of determining if $L(r)$ is empty or not for a single regular expressions. Algorithms to do that can be found in [13].

In the following subsections, we describe algorithms currently used by our ratification process. We note that the categories listed above are by no means exhaustive; they have been derived from our efforts to handle the frequently occurring cases in several practical scenarios. We plan to examine more scenarios and add more categories in the future.

We would also like to note that categories listed above are not mutually exclusive. For example, the expression $5 \leqslant x \leqslant 10, x \in \mathbb{R}$ belongs to both Category 1 and Category 2. Furthermore, by recognizing the equivalence $5 \leqslant x \leqslant 10$ and $(5 \leqslant x) \wedge (x \leqslant 10)$, we can also put this expression in Category 5. This example is somewhat trivial, however it illustrates the point that given a boolean expression, the choice of algorithm to find its satisfiability is not always clear. In our implementation, we use various heuristics to minimize the expected computational burden. In particular, if possible the use of Category 5 is avoided due to its computational complexity[6].

3.1 Domain elimination algorithm

The idea behind the domain elimination algorithm is simple: consider the conjunctive boolean expression $\bigwedge_{i=1}^{n} B_i$, $B_i = x \rhd c_i$, where $c_i, x \in D$ and $\rhd \in \{=, <, \leqslant, >, \geq\}$. Each atomic boolean expression B_i restricts the domain of x to $D_i \subseteq D$.[7] The algorithm works by examining atomic boolean expressions B_i one by one, and by computing $R_l = \bigcap_{i=1}^{l} B_i$ at each step. The algorithm stops either when all inequalities and equalities have been examined, or when $R_l = \phi$. If in the end, $R_n \neq \phi$, then the boolean expression is satisfiable with R_n as its solution.

The fundamental operation performed by the algorithm is the computation of $S_1 \cap S_2$, where $S_1, S_2 \subseteq D$. We achieve

[4]`calendar` data type is used to represent an instance of time. It is similar to the Java Calendar Object or XML dateTime data type.

[5]In our policy language, `composite` data type is used to aggregate several name-value pairs in a single data structure. For the purpose of this discussion, the only relevant property of this data type is the existence of an equivalence class relationship over all `composite` instances.

[6]As discussed later, we have polynomial time bounded algorithms for Categories 1–4, while the complexity for Category 5 is exponential.

[7]We define D to be the universal space corresponding to the data type of x, thus $D = \mathbb{Z}$ when x is integer, and $D = \mathbb{R}$ when x is real, and so forth.

efficiency by observing that there are three fundamental types of domains: totally ordered (ordered with respect to the usual $<$ operation) continuous domains (`real` and `calendar`), totally ordered discrete domains (`integer` and `string`), and discrete unordered domains (`boolean` and `composite`).[8]

First consider the domain elimination for an unordered domain D. The input sets $S_1, S_2 \subset D$ are finite and given by explicitly specifying their elements, that is, $S_i = \{e_1^i, e_2^i, e_3^i, \ldots, e_{n_i}^i\}$ for $i = 1, 2$. Assume that the cost of deleting an element from or inserting an element in a set is negligible compared to the cost of comparing equality between two elements. Clearly, $S_1 \cap S_2$ can be tested in $O(|S_1| + |S_2|)$ using hashing.

For totally ordered discrete and continuous domains, sets S_1 and S_2 can be always expressed in terms of expressions of the following three types: a finite discrete ordered set, an interval, and a union of mutually disjoint intervals. The first two types are sub-cases of the third type. A discrete ordered set has two *extreme boundary points*: the minimum and maximum elements of the set. An interval has two boundary points: the left and right hand side boundary points which may or may not belong to the interval. A union of mutually disjoint l intervals and a discrete set of k elements has $2l + k$ boundary points. It turns out that the set operation $S_1 \cap S_2$ can be performed efficiently by comparing only the (extreme) boundary points of S_1 and S_2. For the most general case, when S_i, for $i = 1, 2$ is a union of mutually disjoint l_i intervals and a discrete set with k_i elements, computing $S_1 \cap S_2$ can be done using a variation of the algorithm to merge two sorted lists resulting in an algorithm of $O(l_1 + l_2 + k_1 + k_2)$.

The domain elimination algorithm can be implemented by using a few domain properties such as total order and whether the domain is continuous or discrete. Thus our implementation is compact, easily maintainable, and largely independent of the data types supported by the policy language. As an added benefit, to support a new data type, all that needs to be done is characterizing the domain properties of the new data type. For example, to add XML date type as a new data type, we would need to say that its domain D is a totally ordered discrete domain and provide a function to compute x^+ given $x \in D$.

Our implementation also covers two generalizations to the type of expressions described above. First, instead of just computing $S_1 \cap S_2$, it is also capable of computing $S_1 \cup S_2$ and S_1^c. The $\cup$ operator is just a union of intervals already considered for single sets, and complements can be done mostly by preprocessing the complements of intervals into a new interval expression without complements. Second, instead of working with just one variable x at a time, it is capable of working with n variables $x_1, x_2, \ldots, x_n$. With these two generalizations, the domain elimination algorithm can find satisfiability and variable assignments for any compound boolean expression whose constituent atomic expressions are of the form $x_i \triangleright c_j$, and thus avoid the use of the solution tree algorithm for such boolean expressions.

[8]A totally ordered domain D is continuous if given $x, y \in D, x < y$, we can always find $z \in D$ such that $x < z < y$. A totally ordered domain D is discrete if for every $x \in D$, either $x = \sup D$ (in case $\sup D$ exists) or there exists $x^+ \in D, x < x^+$, and no $y \in D$, such that $x < y < x^+$.

3.2 Linear inequalities

For boolean expressions in Category 1, we modify Phase 1 of the standard simplex algorithm [9] to determine if there is a non-empty feasible region that satisfies all linear constraints of the atomic boolean expressions. The algorithm consists of the following five steps:

Step 1 Normalize all the linear inequalities and equalities into equations of the form $a_1X_1 + a_2X_2 + \ldots + a_mX_m \triangleright b$ where $\triangleright \in \{=, <, \leqslant, >, \geqslant\}$, $a_j, b \in \mathbb{R}$, and the X_j are variables over $\mathbb{R}$. The variables are lexicographical ordered.

Step 2 Transform all equations into equalities using slack variables and limits. There are five types of equations:

$$\begin{array}{lcl} a_1X_1 + a_2X_2 + \ldots + a_mX_m & = & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & < & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & \leqslant & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & > & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & \geqslant & b \end{array}$$

Each is respectively translated into:

$$\begin{array}{llcl} a_1X_1 + a_2X_2 + \ldots + a_mX_m & & = & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & +S & = & b - \epsilon \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & +S & = & b \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & -S & = & b + \epsilon \\ a_1X_1 + a_2X_2 + \ldots + a_mX_m & -S & = & b \end{array}$$

where S is a new slack variable, $S \geqslant 0$. Note that a new S is introduced for each equality; while ϵ is an infinitesimally small positive constant, the same for all equalities.

Step 3 Build the matrix representation of the system adding an extra column for the constant ϵ. I.e., instead of a single column for the right-hand side of the equalities, the matrix representation has two columns for the right-hand side: one contains the constants b and the other contains the factor multiplying ϵ (initially 0, 1 or -1). The other columns correspond to the variables in the equalities on the left-hand side.

Step 4 Do linear transformations on the matrices with slack variables constrained as $S \geqslant 0$ until finding a feasible solution [9].

Step 5 If a feasible solution does not exist, the spaces do not intersect.

In Step 4, similar to the simplex algorithm, we need to make sure that while transforming the matrix to a row-reduced form and while selecting a column in a row that corresponds to a slack variable, there is no violation of the non-negativity of the slack variables. This is checked by comparing the sign of the factor associated with the slack variable and the sign of the result of adding the two values in columns corresponding to b and ϵ. These two signs must be the same. In addition if one of the factors is 0 the other must also be 0. Let's say that the last column has a constant C. We determining the sign of $b + C\epsilon$ using limit arithmetic:

$$\begin{array}{lcl} b > 0 & \Rightarrow & b + C\epsilon > 0 \\ b = 0, C > 0 & \Rightarrow & b + C\epsilon > 0 \\ b < 0 & \Rightarrow & b + C\epsilon < 0 \\ b = 0, C < 0 & \Rightarrow & b + C\epsilon < 0 \\ b = 0, C = 0 & \Rightarrow & b + C\epsilon = 0 \end{array}$$

If a column corresponding to a slack variable cannot be selected without violating the non-negativity constraint, then the system has no solution. The complexity of this algorithm is $O(n^2 \times m)$, where m is the number of variables and n the number of equations.

Let's run through an example. The following set of equations $\texttt{swap} \geqslant 2\ \texttt{RAM}$, $\texttt{boot} = 1024$, $\texttt{boot} + \texttt{swap} < \frac{1}{4}\ \texttt{HD}$, and $\texttt{RAM} > 0$ is normalized and modified as:

$$\begin{array}{rrrrrcl} 0\,\texttt{boot}+ & 0\,\texttt{HD}- & 2\,\texttt{RAM}+ & \texttt{swap}- & S_1 & = & 0 \\ \texttt{boot}+ & 0\,\texttt{HD}+ & 0\,\texttt{RAM}+ & 0\,\texttt{swap} & & = & 1024 \\ \texttt{boot}- & \frac{1}{4}\,\texttt{HD}+ & 0\,\texttt{RAM}+ & \texttt{swap}+ & S_2 & = & 0-\epsilon \\ 0\,\texttt{boot}+ & 0\,\texttt{HD}+ & \texttt{RAM}+ & 0\,\texttt{swap}- & S_3 & = & 0+\epsilon \end{array}$$

where S_1, S_2, and S_3 are the new slack variables. We now need to find a solution for this system of linear equations under the constraints $S_1 \geqslant 0$, $S_2 \geqslant 0$ and $S_3 \geqslant 0$. We find a solution by doing linear transformations on the equations until we obtain an independent variable for each equation. For our running example, after the linear transformations we obtain the following row-reduced matrix.

$$\left[\begin{array}{rrrrrrrrr} \texttt{boot} & \texttt{HD} & \texttt{RAM} & \texttt{swap} & S_1 & S_2 & S_3 & b & \epsilon \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1024 & 0 \\ 0 & 1 & 0 & 0 & -4 & -4 & -8 & 4096 & 12 \\ 0 & 0 & 0 & 1 & -1 & 0 & -2 & 0 & 2 \end{array}\right]$$

The witness solution is $\texttt{RAM} = \epsilon$, $\texttt{boot} = 1024$, $\texttt{HD} = 4096 + 12\epsilon$, and $\texttt{swap} = 2\epsilon$ with all slack variables, S_1, S_2, S_3, being equal to 0 [9]. Thus, a feasible solution exist, and all linear constraints can be satisfied simultaneously.

Since for two sets A and B, $B \subseteq A$ iff $A^c \cap B = \phi$, the algorithm above can also be used to check if a particular constraint in a set of linear constraints is redundant. In other words, this algorithm can be used to find a dominated policy, and to find whether a set of linear inequalities covers a region defined by another set of linear inequalities.

Finally, we note some limitations of the algorithm given above. First, the algorithm only finds whether a feasible solution exists—it does not find the set of all feasible solutions. Second, the algorithm does not directly handle inequalities of the form $\vec{a}\vec{X} \neq b$ but it can be done by adding the formula $(\vec{a}\vec{X} < b) \vee (\vec{a}\vec{X} > b)$ to the boolean expression and using the solution tree algorithm described in Section 3.3. Last, the algorithm given above does not handle integer-valued variables. We are currently working on modifying the simplex algorithm to address some of these shortcomings [21].

[9]In this particular case, the process of row-reduction did not involve choosing a column corresponding to a slack variable. Therefore, all slack variables can be assigned value 0.

3.3 Solution tree algorithm

The solution tree algorithm is used as the first step in solving satisfiability of a compound boolean expression whose atomic formulae do not fall in the first four categories described in the beginning of this section. In essence solving the satisfiability of such expressions means dealing with $\wedge$, $\vee$ and $\neg$ logical operators simultaneously. Take, for example, the following two conditions, $(X < 10 \vee (X \geqslant 10 \ \wedge\ X + Y < 10))$ and $(X > 12 \vee X > 2Y)$, coming from two different policies. We would like to find out if the conjunction of the formulas $(X < 10 \vee (X \geqslant 10 \wedge X + Y < 10)) \ \wedge\ (X > 12 \vee X > 2Y)$ is satisfiable. If we look at the disjunctive normal form of the formula,

$$\begin{array}{l} (X < 10 \ \wedge\ X > 12) \vee (X < 10 \ \wedge\ X > 2Y) \vee \\ (X \geqslant 10 \ \wedge\ X + Y < 10 \ \wedge\ X > 12) \vee \\ (X \geqslant 10 \ \wedge\ X + Y < 10 \ \wedge\ X > 2Y), \end{array}$$

our problem reduces to checking if at least one of these disjuncts can be satisfied. For this, we can use the domain elimination algorithm to check the first disjunct and solve systems of linear inequalities for the other three.

The solution tree algorithm is based on this intuition, however, it does not explicitly build a disjuncitve form of the input formula. Instead, it takes as an input a boolean expression built with $\wedge$ and $\vee$ connectors, and it returns, one by one, each of the disjuncts on demand. In other words, it works like an iterator: the first time is called, it returns one of the disjuncts that will be passed to the appropriate module for evaluation; next time it is called returns a different disjunct until no more disjunct exits. The order in which the disjuncts are returned is given by a depth-first like traversal of the tree representation of the input boolean expression (the disjunctive normal form above lists the disjuncts in this order). The benefit of using this order is that there is high chance for a disjunct to share some of its parts with the previously evaluated disjunct allowing the reuse of previous computation. In the example, the subsystem $(X \geqslant 10 \ \wedge\ X + Y < 10)$ can be computed in the third disjunct and reused in the fourth. Currently we are using the structure of the formula provided by the users almost unmodified. The only change we do is to push negations down into the atomic boolean expressions.

The algorithm uses two data structures, a stack, `stack`, and a (solution) list, `list`. Initially, `list` is empty. The stack stores nodes from the AND-OR tree representation of the input boolean expression with three *types* of nodes: AND, OR and atomic leaf nodes. AND nodes can be in two states: *non-covered* and *covered*. OR nodes can be in three different states: *non-covered*, *partially covered* and *covered*. Given a node n and one of its ancestors a, we say that a is a *left* ancestor of n if n is part of the left subtree of a; it will be called a *right* ancestor if n is part of the right subtree. We initialize the stack by pushing the root of the tree into the stack. If the root node is an AND or OR node we set its state to non-covered. In addition, if the root is an OR node, the node will have a pointer to the current state of the solution list (in this initial state it points to an empty list). The implementation of the iterator is given by the following algorithm:

```
1. IF $\textsc{top}(\texttt{stack}) = \texttt{null}$: STOP, no more solutions.
2. IF $\textsc{type}(\textsc{top}(\texttt{stack})) = \texttt{atomic}$:
     (a) $N \leftarrow \textsc{pop}(\texttt{stack})$, $\textsc{append}(\texttt{list}, N)$
     (b) $P \leftarrow \textsc{parent}(N)$
           i. IF $\textsc{isRightAncestor}(P, N)$ $\textsc{mark}(P, \texttt{covered})$
          ii. IF $\textsc{type}(P) = \text{OR} \wedge \textsc{isLeftAncestor}(P, N)$:
                $\textsc{mark}(P, \texttt{partial})$
     (c) UNTIL $(P = \texttt{null}) \vee$
         $(\textsc{type}(P) = \text{AND} \wedge \textsc{isLeftAncestor}(P, N))$
         DO $P \leftarrow \textsc{parent}(P)$
     (d) IF $P = \texttt{null}$: RETURN list and STOP[10]
     (e) IF $P \neq \texttt{null}$: $\textsc{mark}(P, \texttt{covered})$,
           $\textsc{push}(P.\texttt{right}, \texttt{stack})$, GOTO 1
3. IF $\textsc{type}(\textsc{top}(\texttt{stack})) = \text{AND} \wedge$
   $\textsc{mark}(\textsc{top}(\texttt{stack})) = \texttt{noncovered}$:
     $\textsc{push}(\textsc{top}(\texttt{stack}).\texttt{left}, \texttt{stack})$.
4. IF $\textsc{type}(\textsc{top}(\texttt{stack})) = \text{OR} \wedge$
   $\textsc{mark}(\textsc{top}(\texttt{stack})) = \texttt{noncovered}$:
     $\textsc{mark}(\textsc{top}(\texttt{stack}), \texttt{partial})$,
     $\textsc{top}(\texttt{stack}).\texttt{backtrackingpoint} \leftarrow \textsc{head}(\texttt{list})$,
     $\textsc{push}(\textsc{top}(\texttt{stack}).\texttt{left}, \texttt{stack})$
5. IF $\textsc{mark}(\textsc{top}(\texttt{stack})) = \texttt{partial}$:
     $\textsc{mark}(\textsc{top}(\texttt{stack}), \texttt{covered})$,
     $\textsc{head}(\texttt{list}) \leftarrow \textsc{top}(\texttt{stack}).\texttt{backtrackingpoint}$,
     $\textsc{push}(\textsc{top}(\texttt{stack}).\texttt{right}, \texttt{stack})$
6. IF $\textsc{mark}(\textsc{top}(\texttt{stack})) = \texttt{covered}$: $\textsc{pop}(\texttt{stack})$.
7. GOTO 1.
```

The complexity of the satisfiability problem is reflected in our algorithm when the disjunctive form of a formula becomes exponentially larger than the original formula. We need to work with heuristics and our approach is to try to find common factors in the disjuncts of the disjunctive normal form to reuse its computation since in our application the evaluation of a disjunct (e.g., solving a set of linear equations) is the most significant portion of the computation. Factoring out subexpressions is also a hard problem. Finding the shortest disjunctive form of a formula is known to be *NP* hard [12]. We currently work with the formula provided by user expecting that the user will write a factored formula. Automatic factorization requires further researh. Other optimization we do is partially evaluating expressions and if the evaluation fails we prune the search. Each time a leaf node is added to the solution list in Step 2a we can immediately check if the partial solution is satisfiable. If so, we proceed as before; otherwise we pop nodes from the stack until we find the first left OR ancestor of the leaf node just added to the solution list (the node causing the pruning) and go back to Step 1. If such an OR node does not exist, there are no more solutions. In our example, if we had $(X \geqslant 10 \wedge X + Y < 10 \wedge Y > 10)$ instead of just $(X \geqslant 10 \wedge X+Y < 10)$, using partial evaluation we would not need to complete the evauluation of $(X \geqslant 10 \wedge X+Y < 10 \wedge Y > 10 \wedge X > 12)$ and we will never attempt to evaluate $(X \geqslant 10 \wedge X + Y < 10 \wedge Y > 10 \wedge X > 2Y)$.

[10]Note that the stack might not be empty, this indicates that there are other solutions.

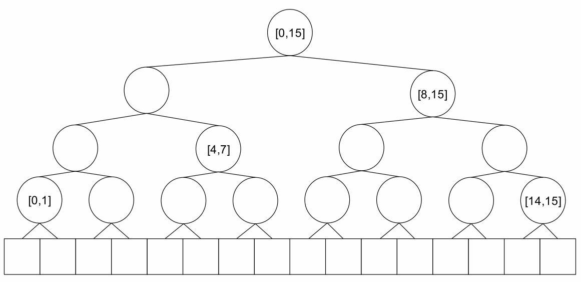


Figure 2. Binary Interval Tree

3.4 Priority assignment algorithm

The priority assignment algorithm is a modified version of an *order-maintenance* algorithm in a list. The basic idea of the algorithm is as follows. Imagine a large array of size 2^n whose i-th cell is marked as occupied if the priority i has already been assigned to a policy. A good choice for n is the word-size of the machine. Logically, we can associate a binary tree of height n with the array so that each leaf of the tree corresponds to a cell in the array (see Figure 2 where $n = 4$). We can also associate a group of consecutive cells to each internal node in the tree. Thus, the root of the tree corresponds to the cells with indices in $[0, 2^n - 1]$, the left child (and the right child, respectively) of the root corresponds to cells with indices in $[0, 2^{n-1} - 1]$ (and $[2^{n-1}, 2^n - 1]$, respectively), and so on. We can also associate a *density* to each internal node that equals the fraction of cells marked as occupied among the group of cells associated with the node. The idea is to keep the density of every internal node below a threshold (< 1) so that there are unoccupied cells close to all locations in the array. In this paper, we set the threshold to be $\frac{1}{2}$.

When a new cell is marked as occupied, we check whether the density of the parent node of the marked cell is above $\frac{1}{2}$. If the threshold is violated, we look for the closest ancestor of the cell where the density is less than $\frac{1}{2}$. We take the cells associated with that node and redistribute the "occupied" mark uniformly in the interval, thus reassigning priorities to some policies while keeping the relative order of the priorities of policies the same. This process ensures that the amortized number of priority reassignment for the policies remains low. The algorithm works as long as the number of policies in the system are less than $\frac{1}{2}$ of the size of the array, which is more than enough for most practical scenarios.

We note, given a cell index (i.e. the priority of a policy) we can find the interval associated with its i-th ancestor node by looking at the binary representation of the index and varying the last i bits. For example, if the index is 23 (10111), the parent interval is [22,23] ([10110,10111]). If the index is 8 (1000) the parent interval is [8,9] ([1000,1001]). To get the grandparent of 23, we vary the last two bits to get [20,23] ([10100,10111]). The interval associated with the grandparent of 8 is [8,11] ([1000,1011]).

The input to our algorithm is an array of size l containing

l policies ordered by their priorities plus the position $i, 0 \leqslant i \leqslant l$ in the array where the new policy needs to be inserted. The core of the algorithm is the following:

Step 1 Insert an empty cell in the input array's i-th position.

Step 2 Let p_{i-1} and p_{i+1} be the priorities of the two policies contained in the cells adjacent to the empty cell. If these priorities are not consecutive ($p_{i+1} - p_{i-1} \neq 1$), then the new policy is assigned a priority p_i in the middle of p_{i-1} and p_{i+1} (e.g., $p_i = \lfloor (p_{i-1} + p_{i+1})/2 \rfloor$), and the policy is inserted in the empty cell.[11]

Step 3 If p_{i-1} and p_{i+1} are consecutive, then we shift priorities for policies that are in the cells $i+1, i+2, \ldots, i+k$ by 1, where k is the smallest integer such that priorities of policies in the cells at position $i+k$ and $i+k+1$ are not consecutive.

If the above step cannot be performed either because the empty cell is in the last position ($i = l$ and $P_{i-1} = 2^n - 1$), or because priorities of policies in cells at positions $i+1$ through l are consecutive up to $2^n - 1$, then we shift priorities for the policies that are in cells $i-1, i-2, \ldots, i-k$ by 1, where k is the smallest integer such that priorities of policies in the cells at position $i-k$ and $i-k-1$ are not consecutive.

Step 4 After the insertion is made, the density of the parent node of the inserted policy is checked. If it is smaller than $\frac{1}{2}$: Stop. Otherwise, find the first ancestor of the inserted policy with density smaller than the threshold and rebalance the priority assignment by evenly redistributing the priorities of policies corresponding to the ancestor found in the previous step.

One important characteristic of the algorithm is that we do not need to build the large binary tree or the cell array explicitly. Thus the storage requirement of our algorithm is proportional to the number of policies in the system l instead of being proportional to 2^n where $l \ll 2^n$. This algorithm guarantees that on average, the amortized reassignment of priorities is $O(1)$ if the number of policies with the same priority is bounded by a constant.

4 Print service management example

We have implemented all four ratification operations described in this paper for an ECA policy system. The core implementation consists of algorithm libraries that are independent of the policy language or model. The ratification module uses these libraries, and is capable of analyzing policies written in ACPL [2]. The module can be invoked from a policy editor to provide feedback to the policy author as she writes new policies, or the module can be invoked by a PDP as new policies are committed into the system [2]. In this section, we will illustrate how the policy ratification module works by using a set of policies defined for managing a print service.

Consider a print service with three queues: low, normal, and high priority queues denoted by $Q_l, Q_n,$ and Q_h, respectively. The print service uses policies to categorize an incoming job into one of these three queues. For categorization, print policies can use the following parameters of an incoming job: number of pages `n`, number of copies `c`, total number of pages currently in the print queues by the same user `N`, and the time of day `time-of-day`. The default policy is to put all printing jobs in the normal queue.

The first set of policies defined by the administrator classifies jobs by the time of day and the number of pages with a goal of expediting printing of small jobs during business hours and to avoid congestion just before the end of the business day:

PL1 If (8 AM $<$ `time-of-day` $<$ 5 PM) $\wedge$ (`n` < 10): `queue` $= Q_h$.
PL2 If (8 AM $<$ `time-of-day` $<$ 5 PM) $\wedge$ ($10 <$ `n` < 30): `queue` $= Q_n$.
PL3 If (8 AM $<$ `time-of-day` $<$ 5 PM) $\wedge$ (`n` > 30): `queue` $= Q_l$.
PL4 If (4 PM $<$ `time-of-day` $<$ 5 PM) $\wedge$ (`n` > 10): `queue` $= Q_l$.

The atomic boolean expressions occurring in the conditional clauses of policies PL1–PL4 are single variable linear inequality constraints. Therefore, the domain-elimination algorithm can be used to analyze policies PL1–PL4. The analysis would alert the administrator to the following facts:

• There are no policies for `time-of-day`=8 AM and 5 PM and `n`=30 and `n`=10 (giving the default normal queue for such jobs regardless of their size or submission time.). In this case, the administrator may want to specify non-strict inequalities for `n` in the policy PL2.

• The policy PL4 may be simultaneously true with policies PL2 and PL3. In this case, the administrator may want to specify that PL4 has higher priority than policies PL2 and PL3.

Figure 3 shows the print service policies in the graphical user interface for our policy tools. This screenshot shows the ratification perspective, where groups of policies can be viewed and analyzed for coverage, dominance, and conflict. The tool shows that PL 2 and PL4 are potentially in conflict as described above.

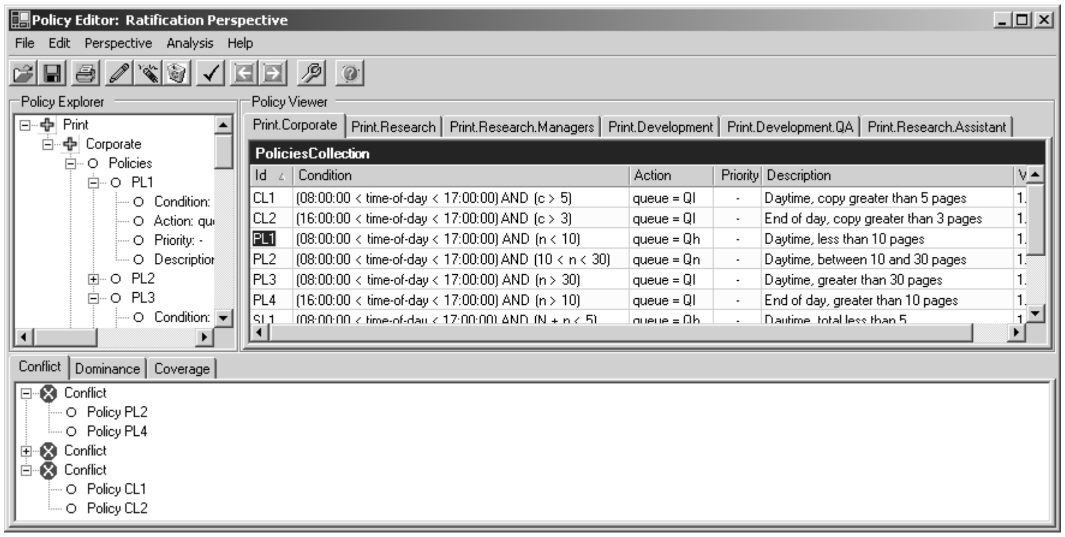
Next, the administrator defines similar policies to give low priority queue to bulk copy printing jobs:

CL1 If (8 AM $<$ `time-of-day` $<$ 5 PM) $\wedge$ (`c` > 5): `queue` $= Q_l$.
CL2 If (4 PM $<$ `time-of-day` $<$ 5 PM) $\wedge$ (`c` > 3): `queue` $= Q_l$.

Both policies CL1 and CL2 can be simultaneously true with policies PL1–PL4. Since CL1 and CL2 assign low priority queues, the administrator may assign both of them a higher priority than policies PL1–PL4. Note that policies CL1 and CL2 can be simultaneously true, however since they specify the same queues, the administrator need not worry about their relative priorities.

Finally, the administrator defines a third set of policies that takes overall current usage of a user into account. These policies make sure that users do not circumvent the previous sets of policies by breaking a large printing jobs into several smaller jobs or that users with multiple printing jobs, but small overall load do not have to wait for larger jobs to

[11] If the empty cell is either the first or the last cell of the array, then there is only one cell adjacent to the empty cell. In such cases, the priority of the missing cell can be thought of as being -1 and 2^n respectively.



Id	Condition	Action	Priority	Description	V
CL1	(08:00:00 < time-of-day < 17:00:00) AND (c > 5)	queue = Ql	-	Daytime, copy greater than 5 pages	1.
CL2	(16:00:00 < time-of-day < 17:00:00) AND (c > 3)	queue = Ql	-	End of day, copy greater than 3 pages	1.
PL1	(08:00:00 < time-of-day < 17:00:00) AND (n < 10)	queue = Qh	-	Daytime, less than 10 pages	1.
PL2	(08:00:00 < time-of-day < 17:00:00) AND (10 < n < 30)	queue = Qn	-	Daytime, between 10 and 30 pages	1.
PL3	(08:00:00 < time-of-day < 17:00:00) AND (n > 30)	queue = Ql	-	Daytime, greater than 30 pages	1.
PL4	(16:00:00 < time-of-day < 17:00:00) AND (n > 10)	queue = Ql	-	End of day, greater than 10 pages	1.
SL1	(08:00:00 < time-of-day < 17:00:00) AND (N + n < 5)	queue = Qh	-	Daytime, total less than 5	1

Conflict
- Policy PL2
- Policy PL4

Conflict

Conflict
- Policy CL1
- Policy CL2

Figure 3. Potential Conflicts in Print Service Policies

finish, etc.

SL1 If (8 AM $<$ `time-of-day` $<$ 5 PM)$\wedge$(N+n < 5): `queue` $= Q_h$.
SL2 If (8 AM $<$ `time-of-day` $<$ 5 PM)$\wedge$(N+n > 60): `queue` $= Q_l$.
SL3 If (4 PM $<$ `time-of-day` $<$ 5 PM)$\wedge$(N+n > 20): `queue` $= Q_l$.

Some of the atomic boolean expressions occurring in the conditional clauses of policies SL1–SL3 involve two variables linear constraints. Therefore, in this case, solution tree algorithm in conjunction with the simplex algorithm would be used to find out that SL1 can be simultaneously true with policies PL1, CL1, and CL2. The policy administrator may assign SL1 a priority equal to PL1, but higher than CL1 and CL2. SL2 and SL3 are punitive policies and the administrator may assign them a higher priority than all the policies that can be simultaneously true with SL2 and SL3.

In the last step of the ratification process, the priority assignment algorithm would take relative priorities given by the system administrator to the policies and assign them numeric priorities consistent with their relative priorities.

5 Related Work

Conflict and anomaly detection have been among the primary research topics in the policy community. In [18], the authors have categorized the conflicts that can occur in SAT policies: (1) modality conflicts, which can be detected without application level knowledge; and (2) goal conflicts, which require outside knowledge.

This categorization has been further developed in [17] where the difference between modality conflicts and consistence conflicts are discussed and possible ways are given to resolve them without human intervention. The main idea of conflict resolution, called domain nesting, is based on the intuition that policies defined for more specific domains are assigned higher priority than the ones for more generic domains. A similar idea has been proposed by [15, 22]. They are based on static conflict detection techniques. We also focus on static analysis but do not confine ourselves to the SAT type policies. Instead, we present common primitive operations that can be used across various policy models such as goal, configuration, and ECA types. Dynamic policy conflict resolution is addressed in [7] but is also based on meta information provided by the policy administrator.

In [4], the authors have studied the anomalies that can happen in a firewall configuration, and present various types of anomalies such as correlation anomaly (similar to policy conflicts in this paper), shadow anomaly (similar to the dominance relation in this paper), generalization anomaly, and redundancy anomaly. While the basic insight on the problem is common, their work is very specific to firewall policies and furthermore specific to their particular implementation of the policy management tool. Thus, their findings cannot be easily applied to other domains.

We have adopted algorithms to maintain order lists under insertion and deletion operations to address our priority maintenance problem. The best upper-bounds known for the order maintenance problem were introduced in [10]. The problem is to keep elements in the right order in a data structure that cannot be modified easily, e.g. a file. The idea behind the algorithms with the best upper-bounds is to leave spaces between elements so that they are minimally reallocated. Part of the concern of these algorithms is to also minimize the extra space required. Our algorithm is based on a revised version of the [10] that uses a simpler data structure, but keeps the same time complexity bounds [6]. Also we do not waste space since we do not move objects we just rename them (i.e. change priorities).

Finding feasible solutions of a set of linear inequalities is a basic step in the Simplex algorithm. We modified the algorithm to take into account the fact that some variables were not bounded and to handle strict inequalities. Non-bounded variables have been studied in [5] for solving problems in graphics applications. That paper also addresses incremental evaluation of linear constraints, a feature required for interactive graphics applications. These techniques apply directly to our backtracking solution tree algorithm.

Non-deterministic algorithms to find solutions to AND-OR trees can be found in many standard AI textbooks. Finding all solutions and pruning algorithms are more sophisticated. Our version is a simplified and adapted version of the backtracking algorithm used by the Warren Abstract Machine to evaluate Prolog programs [3].

6 Conclusions

The first message we would like to convey in this paper is that ratification is a fundamental concept in policy management. It is applicable from high level goal policies to low level Subject/Action/Target or Event/Condition/Action policies. We have further classified ratification into three categories: dominance, conflict, and coverage check. The second message is that useful ratification functions can be developed independently of the domain and policy language. We have built a system that handles ratification for a significant class of policies. To build the system we brought together concepts from constraint satisfaction, linear algebra, and logic programming and designed new algorithms that can be used as building blocks for other management system supporting these three operations. To complete the ratification process, priorities need to be assigned to policies; we also presented an efficient algorithm to do this assignment. We illustrated how the system can be used through a series of examples and showed the policy ratification tool incorporated into our own policy system. We are planning to make ratification tools publicly available so other researchers building policy management systems can incorporate our tools as a library. We hope that other researchers will help expand the tools to cover other classes of policies. For example, the system will benefit if more integer and finite domain constraints are covered. It would also be useful to handle XPath expressions as part of the atomic formulas, and the tools should be expanded so that priorities do not need to be restricted to a total order as for integer priorities. We are studying algorithms that are able to handle partial orders efficiently.

References

[1] D. Agrawal, S. Calo, J. Giles, K.-W. Lee, and D. Verma. Policy management for networked systems and applications. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, May 2005.

[2] D. Agrawal, J. Giles, K.-W. Lee, K. Voruganti, and K. Filali-Adib. Policy-based validation of san configuration. In *Proc. of IEEE Policy 2004*, June 2004.

[3] H. Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991.

[4] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Managemnt*, March 2003.

[5] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[6] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[7] J. Chomicki, J. Lobo, and S. Naqvi. Conflict resolution using logic programming. *IEEE Transactions on Data and Knowledge Engineering*, 15(1):244–249, 2003.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proc. of IEEE Policy 2001*, June 2001.

[9] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[10] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[11] P. Flegkas, P. Trimintzios, G. Pavlou, and A. Liotta. Design and implementation of a policy-based resource managmenet architecture. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Managemnt*, March 2003.

[12] M. R. Garey and D. S. Johnsom. *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman and Company, 1979.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[14] T. Koch, C. Krell, and B. Kramer. Policy defition language for automated managment of distributed system. In *Proc. of Second IEEE Int'l Workshop Systems Management*, pages 55–64, 1996.

[15] M. M. Larrondo-Petrie, E. Gudes, H. Song, and E. B. Fernandez. Security policies in object-oriented databases. In *Proc. of IFIP Database Security: Status and Prospects*, pages 257–268, 1990.

[16] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of American Association for Artificial Intelligence*, July 1999.

[17] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6), November 1999.

[18] J. Moffett and M. Sloman. Policy conflict analysis in distributed system management. 4(1):1–22, 1994.

[19] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy core information model (PCIM) – version 1 specification. IETF RFC 3060, February 2001.

[20] N. Muruganantha and H. Lutfiyya. Policy specification and architecture for quality of service management. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Managemnt*, March 2003.

[21] H. Rueb and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, Computer Science Laboratory, Menlo Park, CA 94025, January 2004.

[22] M. D. Ryan. Defaults in specifications. In *Proc. of IEEE International Symposium on Requirements Engineering*, pages 142–149, 1993.