Security Policy Consistency

Carlos Ribeiro André Zúquete Paulo Ferreira Paulo Guedes
{Carlos.Ribeiro,Andre.Zuquete,Paulo.Ferreira,Paulo.Guedes}@inesc.pt
IST / INESC
Rua Alves Redol N^{o}9 1000 Lisboa
Portugal

Abstract

With the advent of wide security platforms able to express simultaneously all the policies comprising an organization's global security policy, the problem of inconsistencies within security policies become harder and more relevant.

We have defined a tool based on the CHR language which is able to detect several types of inconsistencies within and between security policies and other specifications, namely workflow specifications.

Although the problem of security conflicts has been addressed by several authors, to our knowledge none has addressed the general problem of security inconsistencies, on its several definitions and target specifications.

1 Introduction

Over the years several access control policies have been proposed in the literature. Although these policies cover many different situations and types of information, they are often considered in isolation. Thus, they are not suitable for organizations with complex structures, that manage simultaneously several types of information, thus requiring the simultaneous use of different access control policies. Moreover, policies are often scattered over different environments, making understanding and managing of global policies much more difficult.

Recently there has been a considerable interest in environments that support multiple and complex access control policies, [5, 11, 16, 13]. The goal of these environments is to provide support for the definition of all the policies that makes up the global security policy of an organization into one single platform, thus simplifying management and consistency maintenance.

Some of these environments provide mechanisms to solve potential conflicts between contradictory policies. Some of these mechanisms use special ad-hoc rules to decide upon the acceptability of an action whenever a conflict arises [11]; others use properties such as "authorship", "specificity" and "recency" of security policies to decide on their priority [5, 12]; or combine policies through special operators which decide on the policies' applicability [13].

These mechanisms are used to solve conflicts resulting from the existence of implicit rules in common language. For instance, a user specifies that all his files should not be read by any one else, and simultaneously, he specifies that the files with information about a particular project should be accessible by all members of the project. This situation is not a conflict in common language since the second rule is obviously an exception to the first, but it may be a conflict within a formal security specification.

However, these conflict solving mechanisms should not be used to solve real inconsistencies derived from unification of several policies from several sources. In fact, they can even be detrimental, because they can masquerade real inconsistencies and produce wrong results.

Although conflicts between contradictory policies are the most important type of inconsistency that may be present in a global security policy, they are not the only ones. For instance, a policy may be completely overridden by another policy in such a way that the former policy is completely useless; or the combination of two or more policies may result in a policy that denies every action in the system.

Furthermore, within an organization, it is not enough to verify the security policy self-consistency, it is also necessary to verify the consistency of the security policy with other specifications of the organization. For instance, if an organization's workflow application requires access to some documents and the security policy forbids that access, then the security policy is inconsistent with that workflow specification, which may prevent the organization from working as expected.

In fact, given the constraint nature of security policies, any specification document of an organization which comprises one or more constraints, may be a source of inconsistencies. Thus, the search for security policy inconsistencies does not have a closed solution valid for every organization and situation. Each organization may have different specifications with constraints and different interpretations of security policy inconsistencies.

This paper's contribution is twofold: First, we address the general problem of checking for security policy inconsistencies, whatever they are, on large complex policies; then we address the problem of finding inconsistencies between security policies and other specifications with constraints, namely workflow specifications. Both problems are addressed within a novel approach comprising a tool developed by us (PCV – Policy Consistency Verifier), based on the CHR constraint language [8], which finds inconsistencies within and between security policies and other specifications.

The rest of the paper is organized as follows. We first briefly describe the CHR language. Then describe the tool architecture. Sections 5 and 6 describe how security policy and workflow specifications are handled by the tool. Finally, in section 7 we briefly survey some related work, and in section 8 we conclude the paper.

2 Constraint Handling Rules

CHR is a high-level language designed for writing user-defined constraint systems [8]. CHR is essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR rules are of two types: simplification rules and propagation rules. Simplification rules replace user-defined constraints by simpler ones. Propagation rules add new redundant constraints that may be necessary to do further simplifications.

$$\underbrace{\overbrace{A \le B}^{Constraint}\ , B \ge A}_{Head} \Leftrightarrow \underbrace{true}_{Guard}\ |\ \underbrace{A = B}_{Body}\ \text{//Simplification rule}$$

$$A \le B, B \le C \Rightarrow true \mid A \le C\ \text{// Propagation rule}$$

$$X < Y \setminus X \ne Y \Leftrightarrow true.\ \text{// Simpagation rule}$$

Figure 1: Example of CHR rules.

A CHR rule consists of three parts: a head, a guard and a body (Figure 1). Each part is a conjunction of constraints. There are two kinds of constraints, user-defined and built-in constraints. User-defined constraints are relations that must hold between one or more entities, which assume the form of predicates or operations over those entities (e.g. less(1,2) or 1 < 2). Built-in constraints are simple constraints that can be directly solved by the underlying solver (e.g. A = B).

A simplification rule works by replacing the constraints in the rule's head by the constraints in the rule's body, provided the constraints in the rule's guard are true. A propagation rule adds the constraints in the body but keeps the constraints in the head (Figure 1). Figure 1 shows also a third type of rule named "simpagation". A "simpagation" rule is equivalent to a simplification rule with some of the heads repeated in the body. On a "simpagation" rule only the heads after the "\" sign are removed.

3 Overview of PCV

Security policies can be seen as collections of constraints more or less structured (depending on the security platform used) into complex rules and policies. These constraints may be as simple as “A middle manager cannot approve purchases over a specified amount”, or they can be as complex as constraints comprising forms of prohibition, permission or obligation of user actions.

Given the constraint nature of security policies, the PCV verifier is a natural candidate to be implemented with a constraint language such as the CHR language. This approach simplifies tool construction and potentiates its extensibility to other inconsistency definitions.

PCV is composed by five layers (Figure 2). The first layer is the CHR symbolic solver engine, which is the only layer not comprised of CHR rules. The second layer is composed by the rules which handle basic constraint predicates (e.g. $A \leq B$, $a \in G$) and constitutes the verifier's kernel. The third layer contains the rules which comprise the knowledge on how to decompose the specific constraints placed by each type of specification (security, workflow), into basic constraints. The fourth layer contains rules resulting from the compilation of the different specifications (e.g. security policy specification, workflow specification). Finally, the fifth layer contains the verifier goals, with the definitions of the security inconsistencies being searched.

The purpose of this layering approach is threefold: (i) it simplifies the handler design, because each type of constraints can be handled independently; (ii) it simplifies the proof of correctness, because each layer has no knowledge of the layers on top and see the constraints of lower layers as built-ins; (iii) it simplifies the addition of new specification handlers, by defining the rules required by each specification.

Briefly, the process by which inconsistencies are found works as follows. First the constraints within each specification being verified are compiled into CHR rules. Second, the PCV verifier is invoked with the constraint goals comprising the inconsistency definitions. These goals are successively decomposed into simpler constraints, by the rules generated by the compilers (fourth layer), and by the rules comprising the knowledge on each specification type (third layer), until they can be solved by the kernel rules.

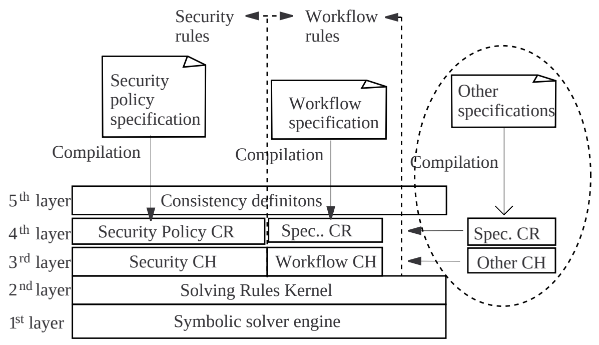

Figure 2: The consistency verifier architecture. CH stands for Constraint Handler, and CR stands for Constraint Rules

In the following sections we proceed by describing each of these layers. For the sake of simplicity, each of the third, fourth and fifth layers were split in two (Figure 2), separating the rules of each layer that handle security constraints from the rules that handle other constraints.

4 Kernel rules

The verifier's kernel rules are responsible for handling basic constraints, resulting from the application of simple operators to basic entities.

Kernel rules are divided into two major groups: rules to handle order and equality operators ($>$, $<$, $\leq$, $\geq$, $=$, $\neq$); and rules to handle membership and set constraints ($\in$, $\notin$, $\cup$ (union), $\cap$ (meet), # (cardinality), [ ] (index)).

4.1 Order and equality rules

The rules to handle order and equality constraints derive from the “minmax” handler proposed by [8], augmented with optional temporal qualifiers but without the “min” and “max” constraints.

A constraint may be timed or timeless. A timed constraint results from applying a temporal qualifier to a timeless constraint (e.g. $X < Y$ becomes $X < Y \texttt{ at } T$). Timed constraints, by opposition to timeless constraints, are only valid at an instant T.

Knowing how to handle timed constraints is of utmost importance for checking for security policy inconsistencies. Many security policies use the notion of time when specifying dependencies from past and future events. For instance, history-based policies like the Chinese Wall policy [7] use the notion of time to allow access to an object only if the same user has not accessed another object in the same class of interest. Another example is given by obligation-based policies which use the notion of time to ensure that some action happens in the future [14].

$$H_1, \dots, H_n \Rightarrow G | B_1, \dots, B_m.$$

derives

$$2^n-1 \begin{cases} H_1 \texttt{ at } T, \dots, H_n & \Rightarrow G | B_1 \texttt{ at } T, \dots, B_m \texttt{ at } T. \\ & \dots \\ H_1, \dots, H_n \texttt{ at } T & \Rightarrow G | B_1 \texttt{ at } T, \dots, B_m \texttt{ at } T. \\ & \dots \\ H_1 \texttt{ at } T, \dots, H_n \texttt{ at } T & \Rightarrow G | B_1 \texttt{ at } T, \dots, B_m \texttt{ at } T. \end{cases}$$

and

$$H'_1, \dots, H'_n \Leftrightarrow G' | B'_1, \dots, B'_m.$$

derives

$$2^n-1 \begin{cases} H'_n, \dots \setminus H'_1 \texttt{ at } T & \Leftrightarrow G' | B'_1 \texttt{ at } T, \dots, B'_m \texttt{ at } T. \\ & \dots \\ H'_1, \dots \setminus H'_n \texttt{ at } T & \Leftrightarrow G | B'_1 \texttt{ at } T, \dots, B'_m \texttt{ at } T. \\ & \dots \\ H'_1 \texttt{ at } T, \dots, H'_n \texttt{ at } T & \Leftrightarrow G | B'_1 \texttt{ at } T, \dots, B'_m \texttt{ at } T. \end{cases}$$

Figure 3: Template rules to build timed propagation and timed simplification rules from their timeless counterparts.

The rules which handle timed constraints derive from the rules which handle timeless constraints in accordance with the templates in Figure 3. A timeless rule with n constraints in its head derive $2^n - 1$ timed rules, each with a different combination of timed and timeless heads. The template rules for timed simplification rules are slightly different from the template rules for timed propagation rules. Timed simplification rules only remove timed constraints. The timeless constraints used to activate each rule are not removed to preserve the activation sequence of timeless rules and therefore maintain their correctness.

built_in	@ $X{=}Y\ at\, T \Leftrightarrow ground(X), ground(Y) \vert X = Y.$
reflexivity	@ $X{=}X\ at\, T \Leftrightarrow true.$
commutativity	@ $X{=}Y\ at\, T \setminus Y{=}X\ at\, T \Leftrightarrow true.$
subsumption	@ $X{=}Y\ at\, T \setminus Y{\leq}X\ at\, T \Leftrightarrow X{\neq}Y \vert true.$
subsumption	@ $X{=}Y\ at\, T \setminus X{\leq}Y\ at\, T \Leftrightarrow X{\neq}Y \vert true.$
irreflexivity	@ $X{=}Y\ at\, T, Y{<}X(at\, T) \Leftrightarrow fail.$
irreflexivity	@ $X{=}Y\ at\, T, X{<}Y(at\, T) \Leftrightarrow fail.$
tautology	@ $X{=}Y\ at\, T, Y{\neq}X(at\, T) \Leftrightarrow fail.$
tautology	@ $X{=}Y\ at\, T, X{\neq}Y(at\, T) \Leftrightarrow fail.$
% Transitivity rules	
WithSelf	@ $X{=}Y\ at\, T, X{=}Z\ at\, T \Rightarrow X{\neq}Y{\neq}Z \vert Y{=}Z\ at\, T.$
WithSelf	@ $X{=}Y\ at\, T, Y{=}Z\ at\, T \Rightarrow X{\neq}Y{\neq}Z \vert X{=}Z\ at\, T.$
WithSelf	@ $X{=}Y\ at\, T, Z{=}X\ at\, T \Rightarrow X{\neq}Y{\neq}Z \vert Y{=}Z\ at\, T.$
WithSelf	@ $X{=}Y\ at\, T, Z{=}Y\ at\, T \Rightarrow X{\neq}Y{\neq}Z \vert X{=}Z\ at\, T.$
WLessOrEqual	@ $X{=}Y\ at\, T, X{\leq}Z(at\, T) \Rightarrow X{\neq}Y{\neq}Z \vert Y{\leq}Z\ at\, T.$
WLessOrEqual	@ $X{=}Y\ at\, T, Y{\leq}Z(at\, T) \Rightarrow X{\neq}Y{\neq}Z \vert X{\leq}Z\ at\, T.$
WLessOrEqual	@ $X{=}Y\ at\, T, Z{\leq}X(at\, T) \Rightarrow X{\neq}Y{\neq}Z \vert Z{\leq}Y\ at\, T.$
WLessOrEqual	@ $X{=}Y\ at\, T, Z{\leq}Y(at\, T) \Rightarrow X{\neq}Y{\neq}Z \vert Z{\leq}X\ at\, T.$
	⋮

Figure 4: Rules to handle timed equality. $(\texttt{at } T)$ stands for an optional time qualification. Although strict CHR rules do not allow optional elements, they are used here to simplify the description.

Although the rules generated by the application of the template rules of Figure 3 are sufficient to handle every timed constraint derived from a user-defined timeless constraint, they cannot handle the timed constraints derived from built-in timeless constraints. While timeless-equality constraints are handled by the underlying built-in solver, the same is not true for timed-equality constraints ($X{=}Y\ at\, T$) which require rules such as the ones in Figure 4. These rules can be divided in two groups: the first is composed of simplification rules describing redundancies and conflicts between timed equality constraints and other constraints; the second consists of propagation rules describing the transitivity properties between timed equality constraints and other constraints.

tautology	@ $X \in G, X \notin G \Rightarrow fail.$
identity	@ $C = A \cap A \Leftrightarrow C = A.$
commutativity	@ $C = A \cap B \setminus C = B \cap A \Leftrightarrow true.$
distributivity	@ $X \in C, C = A \cap B \Rightarrow A \neq B \mid X \in A,\ X \in B.$
revDist	@ $X \in A, X \in B, C = A \cap B \Rightarrow A \neq B \mid X \in C.$
revNotDist	@ $X \notin A, C = A \cap B \Rightarrow X \notin C.$
revNotDist	@ $X \notin B, C = A \cap B \Rightarrow X \notin C.$
notDistrib	@ $X \notin C, X \in A, C = A \cap B \Rightarrow X \notin B.$
notDistrib	@ $X \notin C, X \in B, C = A \cap B \Rightarrow X \notin A.$
revDist	@ $labeling, X \in A, C = A \cap B \Rightarrow A \neq B \mid ((X \notin B, X \notin C); (X \in C, X \in B)).$
revDist	@ $labeling, X \in B, C = A \cap B \Rightarrow A \neq B \mid ((X \notin A, X \notin C); (X \in A, X \in C)).$
notDistrib	@ $labeling, X \notin C, C = A \cap B \Rightarrow A \neq B \mid (X \notin A; X \notin B).$

Figure 5: Basic rules for membership and meet constraints.

4.2 Set and membership rules

Set constraints are not handled as usual. Since the verifier is to be used primarily on non-instantiated specifications, when most sets are yet undefined – in the sense that their members are not yet known– it is not possible to directly solve in order to their contents. Instead of solving *set* constraints, we use them to derive *membership* constraints which can be directly solved. For instance, using the "distributivity" rule, followed by the "tautology" rule of Figure 5, the goal $\langle C = A \cap B, X \in C, X \notin B \rangle$ leads to a *fail* state.

Membership constraints may also be time-qualified such as order constraints. The CHR rules to handle such constraints are also derived from the template rules of Figure 3 but applied to the CHR rules which handle timeless *membership* constraints. Timed *membership* constraints are very useful when sets' contents are dynamic, which happens to be frequent in security policies, e.g. a user may have been playing a role and now he is playing another. On the other hand we do not provide rules to handle timed *set* constraints, since most relations between sets used in security policies (e.g. $C = A \cap B$) are fixed during the whole policy lifetime.

The last three rules of figure 5 contain disjunctions in their bodies (the ';' stands for built-in disjunction), which must be handled by test and backtracking. In order to improve efficiency these rules should be delayed until there are no more constraints in the goal to simplify. The special *labeling* constraint is the last constraint in the goal to be solved. Thus, using this constraint in the head of rules ensures that they are activated only when all other constraints have been already simplified.

Without further assumptions the program composed by the rules in Figure 5 does not terminate, because the constraints generated by propagation rules, may be used to generate other constraints which are going to enable again the same rules. For instance, the constraints derived by the "distributivity" rule could be used by the "revDist" rule to derive the constraint $X \in C$, which may be used again to fire the "distributivity" rule.

However, it is possible to ensure the program termination under three new assumptions:

- The CHR solver verifies the constraint store, before introducing new constraints, to prevent the existence of multiple copies of the same constraint in the constraint store[1].
- Membership constraints are never removed from the constraint store.
- Meet constraints cannot be added to the constraint store, by any program's rule.

Under the first assumption the number of *membership* constraints in the goal store is bounded, provided that the number of variables in the initial goal store is bounded, and that none of the rules derives constraints with new variables. The second assumption is necessary to ensure monotonicity of *membership* constraints in the constraint store. The third assumption is necessary to ensure monotonicity and boundedness of *meet* constraints in the constraint store.

The rules in Figure 6 are used to solve constraints using the restriction operand(:). The restriction operand is a binary operand between a set and a boolean function. The operation defines a new set comprised by all the members of the first set which

[1]Most CHR solvers can be instructed to perform this check by enabling the "already_in_store" option.

$$\begin{array}{ll}
\text{idempotent} & @\ C = A : R(\_) \setminus D = A : R(\_) \Leftrightarrow C = D. \\
 & \\
\text{restriction} & @\ X \in C, C = A : R() \Rightarrow R(X), X \in A. \\
\text{revRestric} & @\ labeling, X \in A, C = A : R() \Rightarrow \\
 & \quad ((R(X), X \in C); (X \notin C, \neg R(X))).
\end{array}$$

Figure 6: Rules for set restriction constraints. The constraint $C = A : R$ is a short form for $C = \{x \in A : R(x)\}$ where C and A are sets and R is a boolean function over x

satisfy the boolean function. The strategy followed by these rules is the same as the one followed by the rules to handle *meet* constraints (Figure 5). The *restriction* constraints are used together with *membership* constraints to derive other *membership* constraints, and the rules containing disjunctions are delayed until the end of the simplification process.

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{\% Relation with other set constraints}} \\
\text{identity} & @\ N1 = |L| \setminus N2 = |L| \Rightarrow N1 = N2. \\
\text{meet} & @\ NC = |C|, NA = |A|, C = A \cap B \Rightarrow NC \leq NA. \\
\text{meet} & @\ NC = |C|, NB = |B|, C = A \cap B \Rightarrow NC \leq NB. \\
\text{join} & @\ NC = |C|, NA = |A|, C = A \cup B \Rightarrow NA \leq NC. \\
\text{join} & @\ NC = |C|, NB = |B|, C = A \cup B \Rightarrow NB \leq NC. \\
\text{restrict} & @\ NA = |A|, NC = |C|, C = A : R \Rightarrow NC \leq NA. \\
\text{less} & N = |A| \Rightarrow integer(N), N < 0 |\ fail. \\
 & \\
\multicolumn{2}{l}{\text{\% For defined sets. Sets with a specific length.}} \\
\text{eqSetMin} & @\ N = |L| \Rightarrow is\_list(L) |\ length(L, N). \\
 & \\
\multicolumn{2}{l}{\text{\% For undefined sets. Sets without a specific length.}} \\
\text{insert} & @\ X \in L, N = |L| \Rightarrow \neg is\_list(L) |\ member(X, L). \\
\text{eqSetMin} & @\ labeling, N = |L| \Rightarrow \\
 & \qquad \neg is\_list(L), integer(N), N \geq 0 | \\
 & \qquad cardinal(L, N). \\
\text{lesser} & @\ labeling, N = |L|, N < N1 \Rightarrow \\
 & \qquad \neg is\_list(L), integer(N1), N1 > 0 | \\
 & \qquad cardinal(L, N1). \\
\text{lesseq} & @\ labeling, N = |L|, N \leq N1 \Rightarrow \\
 & \qquad \neg is\_list(L), integer(N1), N1 \geq 0 | \\
 & \qquad cardinal(L, N1).
\end{array}$$

Figure 7: Rules to handle cardinality constraints. The cardinality of set S is denoted by $|S|$.

The rules to handle cardinality constraints (Figure 7) may be divided into three groups. The first group is used to derive inequality constraints between cardinality values of sets related by *meet*, *union* and *restriction* constraints. The second group is composed of only one rule and is used to translate a cardinality constraint on a defined set into the built-in predicate *length*. This rule is used only if the set is completely defined, in the sense that all its members are known. However, as explained before, most sets are not completely defined at verification time, which makes it impossible to know their cardinality. The third group of rules is used to verify if at the end of the verification process, the number of elements known to be in a set, over which a upper bound cardinality constraint exists does not exceed that upper bound.

5 Security Rules

As described in section 3 the rules which handle specific security constraints are divided into three related layers: the security constraint handler; the security policy rules resulting from the compilation of the security specification; and the consistency definition goals. Each of these layers is dependent on the preceding and following layer and all are dependent on the specificities of the security policy definition environment. The security policy definition environment used in the current prototype is the Security Policy Language (SPL) [13]. This security language was developed by us with the purpose of specifying security controls for complex environments where several security policies must be enabled simultaneously.

In the remaining of this section we briefly describe SPL. Then we describe how the security CH handles the types of constraints placed by SPL. Finally, we describe the process of compiling SPL to CHR using the rules provided by the security CH.

5.1 SPL

SPL is a security language designed to express policies to decide about events' acceptability. An event's acceptability depends on the properties of the event (e.g. author, target and action), on the context at that time and on the properties of past and future events. SPL entities are typed entities with an explicit interface by which their properties can be queried. Some of the entities manipulated by SPL are internal, such as rules and policies, but most are external, like users, files, actions, objects and events. The properties of each external entity depends heavily on the platform (operating system, workflow engine) implementing it.

The language is organized in a hierarchical delegation tree of security policies, in which the master policy is the root delegation starting point. A SPL policy is a structure composed by sets and rules, whose purpose is to express simple concepts like “separation of duty”, “information flow”, or “general access control”.

Sets contain the entities used by the policies to decide on event acceptability. A SPL rule is a function of events that can assume three values: “allow”, “deny” and “notapply”. It’s purpose is to decide on the acceptability of the current event. A rule can be simple or composed. A simple SPL rule is a tuple of two logical expressions. The first logical expression decides on the applicability of the rule, and the second decides on the acceptability of the event.

A SPL rule can be composed by other SPL rules through a specific tri-value algebra with five logic operations: conjunction (AND); disjunction (OR); negation (NOT); existential quantification (EXIST x IN *set rule*); and universal quantification (FORALL x IN *set rule*). These operations behave similarly to their binary homonyms if the “notapply” value is not used and the “allow” and “deny” values are used as true and false, respectively.

Each policy has one special SPL rule called the “query rule” which is identified by a query mark before the name that defines the policy behavior.

```
policy Private( user set OrgUsers ) {
  object set IDocs;                  // Policy data
?Private:                            // Rule name.
  event.action = “SendEmail” &  // Applicability
  event.target IN IDocs              // expression.
  ::                                 // Separation marker.
  event.par[1] IN OrgUsers           // Aceptability
}                                    // expression.
```

Figure 8: Simple policy stating that objects belonging to IDocs can only be sent to users belonging to OrgUsers.

Figure 8 shows a simple policy stating that documents internal to the entity defining the policy cannot be sent to someone outside the organization. The policy has two sets and one SPL rule: the query rule. One of the sets is a policy parameter and contains the users that belong to the organization. The other is internal to the policy and contains the department’s internal documents. The rule uses the special variable “ce” to access the current event properties. The rule’s applicability expression states that the policy is defined only for events whose targets are department’s internal documents internal and whose action is to send an e-mail. The rule’s acceptability expression states that for those events that satisfy the applicability expression the only allowed events are the ones that send the e-mail to a user of the organization.

5.2 Security Constraint Handler

Although most SPL constraints can be handled directly by the verifier’s kernel rules, some cannot. For instance, the constraints resulting from the logical negation of other constraints, or the constraints resulting from using the SPL tri-logical operators over SPL rules, require some additional rules in order to be solved.

$$
\begin{array}{ll}
\text{comutativity} & @\ A \wedge B \setminus B \wedge A \Leftrightarrow true. \\
\text{comutativity} & @\ A \vee B \setminus B \vee A \Leftrightarrow true. \\
\text{comutativity} & @\ A \dot{\vee} B \setminus B \dot{\vee} A \Leftrightarrow true. \\
 & \\
\text{definition} & @\ labeling \setminus A \vee B \Leftrightarrow A{\neq}B|(A;B). \\
\text{definition} & @\ A \wedge B \Leftrightarrow A{\neq}B|A, B. \\
\text{definition} & @\ A \dot{\vee} B \Leftrightarrow A{\neq}B|(A \wedge \neg B) \vee (\neg A \wedge B). \\
\text{identity} & @\ A \wedge A \Leftrightarrow A. \\
\text{identity} & @\ A \vee A \Leftrightarrow A. \\
\text{irreflexivity} & @\ A \dot{\vee} A \Leftrightarrow fail. \\
\text{tautology} & @\ \neg true \Leftrightarrow fail. \\
\text{tautology} & @\ \neg fail \Leftrightarrow true. \\
\text{tautology} & @\ \neg(\neg A) \Leftrightarrow A. \\
\text{deMorgan} & @\ \neg(A \wedge B) \Leftrightarrow (negA \vee negB). \\
\text{deMorgan} & @\ \neg(A \vee B) \Leftrightarrow \neg A, \neg B. \\
\text{definiton} & @\ \neg(A \dot{\vee} B) \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B). \\
 & \\
\text{reduction} & @\ \neg(A < B(\texttt{at}\,T)) \Leftrightarrow B \leq A(\texttt{at}\,T). \\
 & \vdots
\end{array}
$$

Figure 9: Rules to handle logical constraints. The symbol $\dot{\vee}$ stands for exclusive disjunction.

Handling logical negation is straightforward, provided that the verifier also handles logical constraint conjunction and disjunction. The rules which handle these constraints are shown in Figure 9. To simplify SPL to CHR compilation we also provide some rules to handle exclusive disjunction over constraints.

Although logical constraint conjunction and disjunction are handled by direct translation to their built-in counterparts (by the "definition" rules), their negations are handled by the DeMorgan rules, thus pushing negations to basic kernel constraints where they can be handled by the "reduction" rules.

```
definition @ R = not r(D, A) ↔ R = r(D, ¬A).

commutativity @ R3 = R1 and R2 \ R4 = R2 and R1 ⇔ R4 = R3.
identity   @ R3 = R1 and R1 ⇔ R3 = R1.
neutral    @ R3 = r(fail, X) and R2 ⇔ R3 = R2.
neutral    @ R3 = R1 and r(fail, X) ⇔ R3 = R1.
absorb     @ R3 = r(true, fail) and R2 ⇔ R3 = r(true, fail).
absorb     @ R3 = R1 and r(true, fail) ⇔ R3 = r(true, fail).
default    @ R3 = r(true, true)and r(D2, A2) ⇔
                              R3 = r(true, ¬D2 ∨ A2).
default    @ R3 = r(D1, A1)and r(true, true) ⇔
                             R3 = r(true, ¬D1 ∨ A1).
definition @ R3 = r(D1, A1) and r(D2, A2) ⇔
            R3 = r(D1 ∨ D2, (¬D1 ∨ A1) ∧ (¬D2 ∨ A2)).
```

Figure 10: Rules to handle SPL's tri-logical conjunction and negation.

The rules that handle constraints resulting from applying SPL tri-logical operators to SPL rules are straightforward. Most of these rules are just translations of each operator's definition (Figure 10). For instance, the tri-logical conjunction of two SPL rules defined by the predicates $r(D1, A1)$ and $r(D2, A2)$, in which $D1$ and $D2$ stand for the domain expressions of each of the rules and $A1$ and $A2$ stand for the acceptability expressions, is simplified to another SPL rule defined by the predicate $r(D1 \vee D2, (\neg D1 \vee A1) \wedge (\neg D2 \vee A2))$ (definition rule in Figure 10).

The remaining rules reflect special situations in which the result is known without the need to evaluate the definition. For instance, the result of a tri-logical conjunction between two SPL rules in which one of them has an empty domain is equal to the other rule ($R\,\texttt{and}\,r(fail, A)$ is R).

SPL tri-logical quantifiers are slightly more complex to handle. Figure 11 shows the rules to handle the universal quantifier $\texttt{forall}(Set, Tr, R)\,\texttt{at}\,T$, which should be read as $R = \{\forall_x \in Set\,\texttt{at}\,T : Tr(x)\}$. The rules are divided into two sets: the rules that handle universal quantifiers over defined sets; and the rules that handle universal quantifiers over sets

```
% Quantification over proper sets
empty   @ forall(Set, TR, R) at T ⇔ is_list(Set), Set = [ ]|
            R = r(fail, true).
forEach @ forall(Set, TR, R) at T ⇔ is_list(Set),
            Set = [X|Tail]|R = call TR(X), R = R1 and R2,
            forall(Tail, TR, R2) at T.

% Quantification over undefined sets
convert @ forall(Set, Tr, R) at T ⇔ ¬is_list(Set)|
            forall(Set, Tr, R, []) at T.
insert  @ X ∈ Set at T \ forall(Set, Tr, R, U) at T ⇔
            not_member(X, U)|R = Tr(X) and R2,
            forall(Set, Tr, R2, [X|U]) at T.
insert  @ X ∈ Set \ forall(Set, Tr, R, U) at T ⇔
            not_member(X, U)|R = Tr(X) and R2,
            forall(Set, Tr, R2, [X|U]) at T.

no_more@ labeling \ forall(Set, Tr, R, U) at T ⇔
            R = r(fail, true).
```

Figure 11: Rules to handle SPL's tri-logical universal operator.

defined by membership constraints.

Both sets of rules handle the quantifiers constraints by unfolding them to n tri-logical conjunctions. However, the rules that handle quantifiers over undefined sets require an extra constraint property to account for the membership constraints which have already been used with each quantification constraint (Figure 11). This account is necessary to prevent non-termination caused by using each membership constraint more than once with each quantification constraint.

The rules to handle existential quantifier constraints are very similar to the ones that handle universal quantifier constraints. The difference is that existential quantifiers are unfolded to tri-logical disjunctions.

5.3 Compiling SPL

For the purpose of consistency verification, SPL policies should be seen as operator definitions, which are used to state event constraints. The goal of the SPL compiler is to generate the rules necessary to handle each of these types of constraints.

Since SPL is a constraint language, translating it into CHR is a direct process. Each policy is seen as a complex constraint composed by other simpler constraints. Thus, each policy definition is translated into one simplification rule, with one user-defined constraint in the head, several constraints in the body and no guard. The constraint in the head is a predicate with the policy's name, applied to a tuple with a SPL rule representing the policy, every explicit policy's parameters and four implicit ones: current event, global and local variables. The body of the rule is composed by one constraint for each SPL group or rule definition inside the policy. These constraints can then be further simplified by the rule of the consistency engine.

```
private(Event, OrgUsers, Locals, Globals, R) ⇔
        Event  = event(Actor,Action,Target,Pars, Time),
        Locals = locals(private_vars(IDocs)),
        R      = r( Action = “SendMail” ∧ Target ∈ IDocs,
                    Pars[1] ∈ OrgUsers).
```

Figure 12: The figure shows the translation of the SPL policy of Figure 8 to CHR.

Figure 12 shows the translation of the SPL policy presented in figure 8. The policy is translated into a simplification rule with two constraints in the body. One of the constraints states that the set “IDocs” is defined inside the predicate “locals”, in the “privat_vars” section. The other states that the SPL rule to apply is defined by the predicate “r” applied to the tuple composed by the applicability and acceptability expressions.

Although most policies can be translated into a single CHR rule, some require more than one rule, and some require special handling to increase performance. For instance, as referred in the previous section, existential quantifiers are unfolded to tri-logical disjunctions of each of the quantification's elements, which in turn derive built-in disjunctions that may compromise performance due to backtracking. In some situations, existential quantifiers are transformed into conjunctions of two constraints by a process known as Skolemization [10]. This transformation is possible if the set of the quantification is not empty and the applicability expression of the SPL rule does not depend on the quantification variable. Under these conditions, the existential quantification “$\exists_x \in A : rule(x)$” is equivalent to “$c \in A$ and $rule(c)$” where “c” is a Skolem constant.

5.4 Security self-consistency

A security policy may be inconsistent in several ways. For instance, a security policy which is never applicable is unnecessary, thus inconsistent. On the other hand, a security policy that denies every event is also inconsistent. Several other inconsistency definitions may be devised. Currently, our prototype is able to check four types of policy inconsistencies:

- Inapplicability: the policy is never applicable;
- Monotonic denial: the policy denies every event;
- Monotonic acceptance: the policy accepts every event;
- Rule redundancy: one or more rules in the policy are redundant.

Verifying the first three types of inconsistency is straightforward. It is only necessary to find a solution for the *event* variable on each of the following goals:

$$E \in \mathit{AllEvents},\ \mathit{myPolicy}(E, \ldots, r(D, A)),\ D.$$
$$E \in \mathit{AllEvents},\ \mathit{myPolicy}(E, \ldots, r(D, A)),\ \neg D \vee A.$$
$$E \in \mathit{AllEvents},\ \mathit{myPolicy}(E, \ldots, r(D, A)),\ \neg D \vee \neg A.$$

The fourth type is slightly more complex. Briefly, the verifier should replace, in turn, each of the rules to check for redundancy, by a dummy rule with an empty applicability domain, and check for differences between the original policy and the modified policy.

commutativity	@	$R1\,\texttt{diff}\,R2\ \ R2\,\texttt{diff}\,R1 \Leftrightarrow true.$
identity	@	$R\,\texttt{diff}\,R \Leftrightarrow fail.$
definition	@	$r(D1, A1)\,\texttt{diff}\,r(D2, A2) \Leftrightarrow$ $(D1 \dot{\vee} D2) \vee ((D1 \wedge A1) \dot{\vee} (D2 \wedge A2)).$

Figure 13: Rules to handle the `diff` operator. This operator restraints two SPL rules to be different

The replacement of each rule by the dummy rule is done by the underlying platform. The actual test for policy differences is done by the `diff` operator (Figure 13), which restraints the “query” rules of each policy to be different. If this constraint fails, this means that both the original and the modified policies are equal and the replaced rule is redundant.

6 Workflow Rules

Most specifications comprising constraints within an organization are eligible to be checked for consistency together with the organization security policy. The specific importance of workflow specifications results from being usually used all over the organization, potentially crossing different security management domains, thus increasing the probability of occurring inconsistencies.

Although workflow specifications are usually created by a graphic tool, and kept inside a workflow framework in an internal format, several workflow frameworks also support the "Workflow Process Description Language" (WPDL) [15] for specification interchange purposes. Given the interchange purpose of WPDL we have found it to be the ideal language to test the verifier's ability to express and handle workflow specifications.

In this section we briefly describe WPDL. Then we describe how WPDL is translated into CHR, and why the workflow CH for WPDL is empty. Finally, we give some examples of cross-consistency goal definitions.

6.1 WPDL

WPDL's main entities are: *activities*, *participants* and *transitions*. Each activity is a logical, self-contained unit of work within the workflow definition, performed by a participant. An activity may be atomic, a sub-flow, loop activity or a dummy activity.

Atomic activities are the ones which are going to be activated by events and therefore controlled by the security policy. A sub-flow activity is just a container for a sequence of activities. A loop activity is a special control activity comprising a loop condition. For each loop activity there are two special transition entities pointing from and to it. The outgoing transition points to the loop's start activity. The incoming transition points from the loop's end activity. Dummy activities are used to support routing decisions within the incoming and/or the outgoing transitions.

Transitions are comprised by three elementary properties: the from-activity, the to-activity, and the activation condition. Transitions execution may lead to sequential or parallel activities execution. The information related to split and join properties is defined within the appropriate activity.

The join property decides if the activity requires the activation of only one or every incoming transition. The split property decides if, after the activity is performed, every outgoing transition is activated or if only one of them is activated. In the latter case the activated transition is chosen from a priority list. If the first transition cannot be activated due to its activation condition, the next transition is chosen.

Participants are resources that may be assigned to activities. A participant may be a person, a role, an application (automated activity), or an organizational unit.

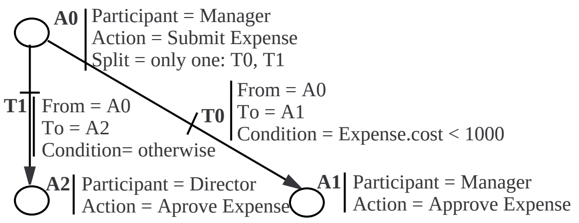

Figure 14: A workflow definition example: A_i stands for activity i and T_i stands for transition i.

Figure 14 shows an example of a workflow definition. "Activity 0" is performed by a Manager and is an expense submission. After being executed, "activity 1" enables two transitions, but only one of them can be executed. It first tries "transition 0", if the expense's cost is less than 1000 then "transition 0" is executed and "activity 1" becomes enabled, otherwise "transition 1" is executed and "activity 2" becomes enabled.

6.2 Compiling WPDL

For the purpose of consistency verification, each WPDL activity specification is seen as the definition of a specific type of logical operator, to state event constraints. These constraints fail when the event is incompatible with the activity. For instance, when the author of the event is different from the participant required by the activity.

Transitions may also be seen as logical event operators. The purpose of these operators is to establish temporal constraints between events enacting the to- and the from- activities of the transition.

The WPDL compiler generates the CHR rules for handling these constraints. However, due to the WPDL's simplicity it is not necessary to provide auxiliary CHR rules, such as the ones comprising the security constraint handler (section 5.2). Thus the specific workflow constraint handler for WPDL is empty.

```
a0(E, Globals) ⇔
    E = event(Actor,Action,Target,_,_), E ∈ AllEvents,
    Actor ∈ Clerk, Action = "Build", Target = Budget.

a1(E, Globals) ⇔ t0(E, Globals),
    E = event(Actor,Action,Target,_,_), E ∈ AllEvents,
    Actor ∈ Clerk, Action = "Approve", Target = Budget.

a2(E, Globals) ⇔ t1(E,Globals),
    E = event(Actor,Action,Target,_,_), E ∈ AllEvents,
    Actor ∈ Boss, Action = "Approve", Target = Budget.

t0_test(E, Globals) ⇔ Budget = budget(Cost), Cost < 1000.
t0(E, Globals) ⇔ t0_test(E,Globals), a0(PreviousE,Globals),
    E = event(_,_,_,_, T), E ∈ AllEvents,
    PreviousE = event(_,_,_,_, PreviousT), PreviousT < T.

t1_test(E, Globals) ⇔ ¬ t0_test(E, Globals).
t1(E, Globals) ⇔ t1_test(E, Globals), a0(PreviousE,Globals),
    E = event(_,_,_,_, T), E ∈ AllEvents,
    PreviousE = event(_,_,_,_, PreviousT), PreviousT < T.
```

Figure 15: This figure shows the translation of the workflow of Figure 14 to CHR. AllEvents, Clerk, Budget and Boss variables are defined within Globals (not shown for simplicity)

Figure 15 shows the rules resulting from the compilation of the workflow definition of Figure 14. Activity constraints are simplified to conjunctions of basic constraints on event properties and of transition constraints.

Transition constraints are simplified to basic constraints and activity constraints. The actual constraint simplification of transition constraints is divided into two simplification rules, to assist in expressing dependency on other transitions, when they are in the same split priority list. For instance, the constraints of type $t1$ depend on the failure of the test condition of constraints of type $t0$.

6.3 Workflow/Security consistency

A workflow specification may be inconsistent with a security policy in several ways. For instance, a workflow may be inconsistent if at least one of its activities cannot be executed under the security policy. It may also be inconsistent if there is no activation path between its start activity and its end activity allowed by the security policy.

The last situation is particularly interesting since it reflects the impossibility of performing the work for which the workflow was conceived. To verify this inconsistency it is only necessary to find a solution for the event variable E which satisfies the goal:

$$E \in \mathit{AllEvents},\ \mathit{lastActivity}(E,\dots),$$
$$\texttt{forall}(\mathit{AllEvents}, Tr(\dots), R).$$

where

$$Tr(E,\dots,R) \vdash \mathit{masterPolicy}(E,\dots,R), \mathit{close}(R).$$

The first line of the above goal states that there is an event, belonging to the events set, which satisfies the last activity constraints. Since, by definition, the last activity implies the existence of an event for each of the preceding activities, the goal states the existence of an event for each activity from the start activity to the end activity. The second line of the consistency goal states that every event must also satisfy the security policy, thus stating consistency between the two specifications.

The "close" constraint is an auxiliary constraint which specifies the behaviour of the security service when a security policy does not apply to an event. With the close assumption, the service denies those events. With the open assumption, the service allows those events. The rules to handle the "close" and "open" constraints are defined as:

$$\texttt{open}\, r(D1, A1) \Leftrightarrow \neg D1 \vee (D1 \wedge A1).$$
$$\texttt{close}\, r(D1, A1) \Leftrightarrow D1 \wedge A1.$$

These rules have another important function. They act as the bridge between the tri-value logic used by the security constraints and the binary logic used by the workflow constraints.

Although we have not exhaustively tested with many different inconsistency types, the results we have obtained so far and the flexibility of the underlying platform, lead us to believe that PCV is able to find most types of inconsistencies within and between security policies and other specifications.

7 Related work

The security inconsistency problem has been addressed by many authors. Some solve conflicts within security specifications by adding implicit rules to incomplete specifications [11, 5, 12]. Others detect inconsistencies in security properties within workflow specifications [6, 4].

Jajodia *et al* [11] define a logical language with ten predicate symbols to express security policies. Three of them are authorization predicates (dercando, cando, do), used to define the allowed actions. Although not explicitly, these predicates define three authorization levels with "dercando" as the weakest and "do" as the strongest. The "do" predicates are used to solve conflicts between "cando" predicates. "dercando" predicates are derived from "cando" predicates, and are overridden by "cando" predicates when in conflict.

Another approach to conflict resolution, presented in [5] and [12] uses elements such as rule authorship authority, rule specificity and rule recency to prioritize rules. Although simple and natural, this approach may lead to undesired behavior. It is not uncommon for a high authority manager to issue a rule which may be overridden by a low authority manager, or to express a mandatory general rule which should not be overridden.

Bertino *et al* [6] also use constraints to detect inconsistencies of roles assignment to workflow tasks, and to plan effective inconsistent-free role assignments. Although the work is able to model several types of restrictions on role assignment, namely several forms of separation of duty, it does not consider any other kind of security or workflow restrictions.

Atluri and Huang [4] use a different approach to detect inconsistencies between security and workflow specifications. They model security and workflow restrictions as Petri nets, and state that the safety problem in the authorization model is equivalent to the reachability problem in that type of nets. They assume a model where authorization restrictions are the subset of workflow restrictions that specify users and roles authorizations.

However, to our knowledge, the general problem of inconsistency detection on complex security policies, comprising several types of inconsistency, including inconsistencies with other specifications, has never been addressed by any author.

8 Conclusions

We have defined a tool (PCV) to detect inconsistencies within security policies and between security policies and workflow specifications. PCV is able to detect several inconsistency types within security policies defined with the SPL language, which is able to express complex security polices, and between those security policies and WPDL workflow specifications.

PCV is easily adapted to each organization's needs by allowing the definition of other inconsistency types and target specifications.

Currently, our prototype has approximately 300 CHR rules running over the CHR solver provided with SICstus Prolog [3], and is able to handle all SPL and WPDL constraints. Some experiences have been performed using compositions of SPL policies and workflow specifications to validate the process. Namely, we have tested several workflow specifications with security policies comprising separation duty, information flow, and other types of security policies, with promising results.

This work is part of a security platform which also includes a security language able to simultaneously express several complex security policies –the SPL language– and a compiler which is able to enforce it efficiently within an event monitor service.

Acknowledgments

We thank David Matos for reviewing earlier of this article. We also thank the reviewers of CL2000 workshop on Rule-Based Constraint Reasoning and Programming for pointing out some problems on early

version of this article.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. *Lecture Notes in Computer Science*, 1330:252–267, 1997.

[2] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 1999.

[3] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjöland, and J. Widn. SICStus prolog user's manual. Technical Report T93-01, Swedish Institute of Computer Science, Oct. 1995.

[4] V. Atluri and W.-K. Huang. An authorization model for workflows. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth ESORICS*, LNCS, pages 44–64, Rome, Italy, Sept. 1996. SV.

[5] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proceeding of the 12th IEEE Computer Security Workshop*. IEEE Computer Society Press, July 1999.

[6] E. Bertino, E. Ferrari, and V. Alturi. A flexible model for the specification and enforcement of role-based authorizations in workfiow management systems. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 1–12, New York, Nov. 6–7 1997. ACM Press.

[7] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *SympSecPr*, Research in Security and Privacy, pages 206–214, Oakland, CA, May 1989. IEEE, IEEECSP.

[8] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, Oct. 1998.

[9] T. Frühwirth. Termination of chr constraint solvers. In *CompulogNet Area on Constraint Programming Workshop*, Paphos, Cyprus, Oct. 1999. ERCIM Working Group on Constraints.

[10] C. J. Hogger. *Essential of Logic Programming*. Oxford University Press, 1990.

[11] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):474–485, June 1997.

[12] N. Li, J. Feigenbaum, and B. N. Grosof. A logic-based knowledge representation for authorization with delegation. In *Proceeding of the 12th IEEE Computer Security Workshop*. IEEE Computer Society Press, July 1999.

[13] C. Ribeiro and P. Guedes. Spl: An access control language for security policies with complex constraints. Technical Report RT/0001/99, INESC, Jan. 1999.

[14] C. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes. Enforcing obligation with security monitors. Technical report, INESC, Apr. 2000.

[15] WFMC. Interface 1: Process Definition Interchange Process Model (WFMC-TC-1016-P). Technical report, Workflow Management Coalition, Brussels, Nov. 1998.

[16] T. Y. C. Woo and S. S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 1992 IEEE Computer Society Symposium on Security and Privacy (SSP '92)*, pages 33–51. IEEE, May 1992.