Synthesising verified access control systems through model checking

Nan Zhang [a], Mark Ryan [a,*] and Dimitar P. Guelev [b]

[a] *School of Computer Science, University of Birmingham*
E-mail: {nxz,mdr}@cs.bham.ac.uk
[b] *Institute of Mathematics and Informatics, Bulgarian Academy of Sciences*
E-mail: gelevdp@math.bas.bg

We present a framework for evaluating and generating access control policies. The framework contains a modelling formalism called RW, which is supported by a model checking tool. RW is designed for modelling access control policies, and verifying their properties. The RW language is very expressive, allowing us to model complex access conditions which can depend on data values, other permissions, and agent roles.

A property expresses the capability of a coalition of agents to achieve a goal, which may include reading and overwriting certain information. Given a model built based on a policy and a property, the model-checking algorithm decides whether the goal defined by the property is achievable by the coalition within the permissions the policy provides. In the case that the goal is achievable, the algorithm outputs strategies which may be used by the coalition to achieve the goal.

The unachievability of legitimate goals may suggest that the policy does not provide the users enough permissions to carry out their actions. The achievability of malicious goals may reveal certain security holes in the policy. When malicious goals are achievable, the resulting strategies help to provide clues on amending the policy. The tool implements the algorithm and thus performs the RW model-checking. It can also convert a policy written in the RW language into a policy file in XACML. An access control system can then be built on the converted policy file.

1. Introduction

The importance of access control is growing rapidly in a world where computers are ever-more interconnected. Access control policies are authorisation strategies upon which access control systems are built. For an access control system to be effective, it is important to ensure that its access control policy is properly defined. The policy of an access control system must fit the services that the system intends to provide. The fitness of an access control policy consist of two aspects. First, the policy should provide users enough permissions to carry out their actions and achieve their

[*] Corresponding author: Mark Ryan, School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK. Tel.: +44 121 414 7361; Fax: +44 121 414 4281. E-mail: M.D.Ryan@cs.bham.ac.uk

legitimate goals. Secondly, at the same time, the policy should prohibit malicious goals from being reached.

Modern researchers tend to use tool-based approaches to evaluate access control policies, because the complexity of access control policies often makes reasoning about them by hand infeasible. Fisler, Krishnamurthi, Meyerovich and Tschantz [14] have created Margrave, a software suite for analysing role based access control policies. Halpern and Weissman [19] have demonstrated how a fragment of first-order logic can be used to represent and reason about access control policies. Schaad and Moffett [29] used Alloy [22,23] to specify a RBAC-style model [28], ARBAC97-style extensions [27] and a set of separation of duty properties [26].

However, what has long been neglected by much previous work is the analysis and detection of security holes in policies caused by *interactions of rules*, *co-operations between agents* and *multi-step actions*. It is not enough to know:

- whether a single rule behaves correctly, but that all rules, working together, behave correctly;
- what a single agent can do by herself, but what a set of agents can achieve through co-operations, including perhaps overwriting each other's privileges;
- what an agent can do in a single action, but what he can achieve through a sequence of actions, especially when agents can change permissions to give themselves or others privileges.

To see the importance of these observations, consider the following example.

EXAMPLE 1.1. Consider a *conference paper review system*. It consists of a set of agents which are programme-committee (PC) members and a set of papers to be reviewed by the PC members. The following rules apply:

1. The chair of the PC assigns papers to PC members for reviewing.
2. PC member a can read PC member b's review for a paper p provided p is not assigned to a.
3. If two PC members, a and b, are both assigned paper p for reviewing, a can read b's review for p only if a has already submitted its own review for p. This is also true for b.
4. Having been assigned a paper p, PC member a can give up being reviewer for p before the reviewing is finished.

The purpose of having these rules is to prevent a reviewer's opinion on a paper from influencing another reviewer's. Although each of these rules seems to be sound, the intention of rule 3 can be easily breached by several agents working together through multiple steps of actions. Given a paper p, three PC members a, b, and c, with c being the chair, see the following two strategies available for a and c to work together to breach the intention of rule 3.

STRATEGY 1.1. (1) c assigns p to b for reviewing (permitted by rule 1).

(2) a reads b's review for p (permitted by rule 2).
(3) c assigns p to a for reviewing (permitted by rule 1).

STRATEGY 1.2. (1) c assigns p to both a and b for reviewing (permitted by rule 1).
(2) Before a submits his review for p, he resigns as reviewer for p (permitted by rule 4).
(3) a reads b's review for p (permitted by rule 2).
(4) c assigns p to a for reviewing again (permitted by rule 1).

Each single step of the above two strategies is legitimate. However, the strategies enable a behaviour which was not intended to be permitted by the rules. Three reasons have caused this problem:

1. *Interactions of rules*. Although rule 3 explicitly prohibits a reviewer from reading another reviewer's review for the same paper assigned to both of them, rule 2 and rule 4 provide opportunities for the agents to by-pass it.
2. *Co-operations between agents*. Although a cannot breach rule 3 by himself, with the help of c, they can act together through an indirect way to get around of rule 3. This co-operation involves the consignment of the privilege of reviewing p to a by c.
3. *Multi-step actions*. Although a cannot breach rule 3 in a single step, he, with the help of c, can violate it following a sequence of actions.

Generally speaking, Example 1.1 shows how a set of naively designed access control policy can exhibit security holes. Such holes typically cannot be identified by the imagination of designers; scarcely can they be detected by most of traditional approaches, such as static analysis [2,10] and policy-querying [14].

The framework presented in this paper includes a tool which uses a *model-checking* algorithm to evaluate the fitness of access control policies. Given an access control policy, the tool builds a model, M, based on the rules of the policy. It then uses the model-checking algorithm to check whether M preserves certain security properties. Using model-checking to evaluate access control policy has a number of advantages over other approaches:

- Because a model, M, is built based on the policy, it enables us to understand the policy as a whole. Any change made on a single rule or caused by adding/deleting a rule will be reflected on M. This makes the study of interactions of rules easier.
- Model-checking's ability to perform temporal reasoning is suitable for exploring possible consequences of multi-step actions.

Along with the advantages of model-checking, the algorithm considers coalition of agents instead of a single agent when the checking is performed. The modelling formalism that we use (the RW formalism) considers permissions as data that are subject to permissions. This consideration makes it possible to explore the consequences of changing of the agents' privileges. Therefore, potential attacks caused by co-operations between agents may also be uncovered.

The framework consists of:

The RW access control formalism. This is our modelling formalism, which is used to model access control policies. This formalism is based on propositional logic. A novelty of this formalism is its built-in abilities to express permissions about permissions (sometimes known as meta-policy [7]). The RW (Read and Write) formalism considers permissions as data the same way as it considers ordinary data in a system. Thus, permissions are objects of reading and writing actions just as other ordinary data are.

The RW access control policy description and specification language. This is a machine-readable language which is used to express access control policies modelled in the RW formalism and properties to be verified against the model. A property is a query, asking, given a set of agents and a goal, whether the agents can achieve the goal by carrying out strategies consisting of permissible reading and overwriting actions in each step. Goals amount to either learning about the state of the system to which the policy applies, or changing it to satisfy certain conditions, or some logical combinations of these.

The RW model-checking algorithm. This algorithm takes a model of a policy and a property as input and answers whether the property holds on the model. The algorithm uses the technique of symbolic model-checking [25]. If the property holds, which means the agents can achieve the goal, the algorithm outputs strategies that may be used by the agents to achieve the goal. For legitimate goals, the achievability shows that the policy provides enough permissions to the users. However, for malicious goals, the achievability may reveal certain weaknesses in the policy. In these cases, the strategies that are output provide clues regarding to how to amend the policy.

AcPeg. This is a tool written in Java, which implements the above algorithm to perform the checking. The tool can be obtained from [33]. It can also translate the policy-description in the RW language into a policy file in XACML [15]. The policy file in XACML can then be used to implement a real access control system. A relational database, which is assumed to contain the access-control-relevant data of the system, must be set up for helping to make access decisions based on the translated XACML policy file.

The RW formalism, the mathematical form of the description part of the RW language and a decision procedure are presented in [16]. Given a set of access control policy, a goal, and a set of agents, the decision procedure can figure out whether there are strategies available for the agents to achieve the goal, however, without demonstrating what the strategies are. The algorithm presented in Section 5 is developed on the basis of that decision procedure in order to extract the strategies. In the cases of malicious goals, the resulting strategies may give us clues that how the goals can be achieved and thus how the policy can be amended accordingly. The algorithm, the specification part of the RW language, and AcPeg's model-checking ability are presented in [35]. The description-part of the RW language in machine-readable form, AcPeg's translating ability and related issues are presented in [34]. The current paper integrates the contents of the three papers plus some new developments and results that have not been presented before.

Structure of the paper

Related work is discussed in Section 2. The formal definition of the RW formalism is introduced in Section 3 as well as the description part of the RW language. How to write properties in the RW language is explained in Section 4. The RW model-checking algorithm is presented in Section 5. The implementation of the algorithm and experimental results are discussed in Section 6. The translation from RW to XACML is explained in Section 7. Conclusions are drawn in Section 8.

2. Related work

Automatic analysis of access control policies has attracted much attention of researchers in recent years. Throughout the rest of this section, we discuss some current approaches and compare them with our work. We also explain why the widely-used verficiation tools – Alloy, SMV and Mocha – are not suitable for our work, and as a result the creation of our own tool becomes necessary.

Fisler, Krishnamurthi, Meyerovich and Tschantz [14] focus on verification and change-impact analysis of role-based access control policies written in XACML. They have a tool called Margrave, which reads XACML, translating them into multi-terminal decision diagrams (MTBDDs) to answer queries. MTBDDs are a more general form of BDDs. Unlike a BDD which only has two terminals, 0 and 1, a MTBDD can have a set of terminals. Because XACML policy evaluation may lead to the result of *permit*, *deny* and *not-applicable*, MTBDDs are more suitable for translating XACML policies than BDDs. Margrave verifies whether a rule preserves a property by taking a query which expresses the property as input and output the answer to the query. It does not perform model-checking. It simply traverses the MTBDD which represents the rule, using the information provided in the query and seeing which terminal it gets to.

Change-impact analysis is also an important aspect of their work. Margrave can take two rules that span a set of changes as input, and output a summary of the differences.

Two big advantages of their approach are *performance* and *scalability*. According to their experimental data, most of verification tasks take no longer than 10 milliseconds (ms), however converting rules to MTBDD form takes from 70 ms to 335 ms.

Memory consumption is about 4.7 Mbytes. Because MTBDDs scale up quite well, their tool is quite capable to handle large cases.

However, the problems raised by interactions of rules, multi-step actions and co-operations between agents are not mentioned in their work.

Schaad and Moffett [29] demonstrate how to use Alloy to check that separation-of-duty constraints may be breached when policies are changed by administrative policies defined in the ARBAC97 model. We also had considered using Alloy as our modelling formalism and the Alloy analyser as our checking tool. However, because Alloy does little temporal reasoning, if we use Alloy, we have to hard code system states and transition relations explicitly by ourselves. From our experience, we found that this makes a model in Alloy very complex and the checking very slow. Alloy's lacking of temporal reasoning makes it unsuitable for our work.

Guttman, Herzog, Ramsdell and Skorupka [18] present a systemic way to analyse access control policies in the Security-Enhanced Linux system (SELinux). They develop a highly abstract model of the SELinux operating system access control mechanism. In this model, the system configuration determines a transition system which represents possible information flows. Properties of the system take the form of information flew security goal statements which describe the objectives that SELinux is intended to achieve. The goal statements are written in LTL [13]. They have tools which take the transition system and produce input for NuSMV [8]. The analysis then can be done by model checking. Their approach is also known as *rigorous automated security management*, which also applies to network security management [17].

We had considered using the input language of SMV to express access control policies, CTL [20] to specify properties and SMV to model-check. This seemed possible to us because in the RW formalism, resources and various relations in an access control system are represented by propositional variables. These propositional variables can all be defined as boolean variables in SMV. The abilities of an agent to read and write a resource can be represented by two boolean variables. The conditions defining the situations under which the resource can be accessed can be defined as logical formulas which update the boolean variables' values. The property we want to check is reachability; that is, given a goal and a group of agents, whether there are strategies available for the agents to achieve the goal. This sense of reachability may also be expressed by CTL formulas. SMV's ability of temporal reasoning also seems suitable for the checking of reachability. However, despite all these similarities, we cannot use SMV. The strategies we are looking for consist of *overwriting* steps and *sampling* steps. An overwriting step produces only one possible outcome, while a sampling step produces two possible outcomes, which can be $\top$ or $\bot$, according to the value of the variable being sampled. Each of the two outcomes becomes an origin of a new path extending to the future. For a strategy to work, the goal has to succeed on all paths resulting from the user choices no matter the sampling outcome is $\top$ or $\bot$. CTL only has the universal quantifier A and the existential quantifier E over paths. It cannot be used to express the idea that a goal succeeds on all paths satisfying the user's choices.

Since our approach of model-checking involves considering activities of agents, it seems that ATL [4] and its model checker Mocha [3] are good candidates for our work. However, we had to give up the idea of using them. We have experience of using ATL and Mocha to model-check access control policies [32]. From that experience, we learned that using Mocha to analyse big systems is very slow. Moreover, a program in Mocha is composed of one or more than one module. A module defines a number of variables and several atoms. Each atom declares variables that it controls, reads and awaits. It is natural to use a module to represent the behaviors of the policies of an access control system with atoms in the module to represent agents' activities permitted by the policies. However, the problem is that, in a Mocha program, each variable defined in a module can only be controlled by at most one atom, which means only one atom has the ability to change its value. This constraint implies that if we use the Mocha program to represent an access control system, the value of each variable in the system can only be modified by one agent. In practice, we found this is very inconvenient.

3. The RW formalism and its description language

3.1. Definition

DEFINITION 3.1. Let $\mathbf{L}(P)$ be the set of the propositional logic formulas built from the propositional variables in the set P. An access control system S is a tuple $\langle \Sigma, P, \mathtt{r}, \mathtt{w} \rangle$, where Σ is a set of *agents*, P is a set of *propositional variables* and the mappings $\mathtt{r}, \mathtt{w} : P \times \mathcal{P}(\Sigma) \to \mathbf{L}(P)$ specify the immediate access rights of coalitions of agents.

According to this definition, states of S are valuations of the variables in P. An agent $a \in \Sigma$ is allowed to read and overwrite variable p iff the current state of the system s satisfies the propositional logic formula $\mathtt{r}(p, \{a\})$ and $\mathtt{w}(p, \{a\})$, respectively. We assume that access rights are exercised by one agent at a time in this paper for the sake of simplicity and write $\mathtt{r}(p, a)$ instead of $\mathtt{r}(p, \{a\})$, and similarly for $\mathtt{w}$. Thus the formulas $\mathtt{r}(p, a), \mathtt{w}(p, a) \in \mathbf{L}(P)$ define the conditions under which a is permitted to read and overwrite the value of p. They are functions on S's states.

3.2. Example – a conference paper review system

The example access control system is an extension of the *conference paper review system* introduced in Example 1.1.

EXAMPLE 3.1. A conference paper review system includes fixed sets of agents and papers. Some of the agents are authors of the papers and/or participate in the conference Programme Committee (PC). The PC has a chair. Rules in the policy which applies to this system include:

1. Whether an agent is a PC member, or is the chair, or is an author of a paper is readable by all the agents. Authorship of papers cannot be changed.
2. The PC chair appoints agents to be PC members. A PC member can resign her membership.
3. The PC chair can assign a paper to a PC member for reviewing, provided the PC member is not the paper's author.
4. Whether a PC member is a reviewer of a paper is readable by all the PC members except the author(s) of the paper.
5. A reviewer of a paper can assign the paper to be sub-reviewed by another agent who is not an author of the paper and has not been assigned the same paper by another reviewer.
6. A reviewer of a paper can give up being reviewer, unless he has already appointed a sub-reviewer for the paper.
7. Whether a PC member is a sub-reviewer of a paper is readable by all the PC members except the author(s) of the paper.
8. A sub-reviewer of a paper can give up being sub-reviewer, unless he has already submitted the review for the paper.
9. Whether a review for a paper has been submitted is readable by all the PC members, except the author(s) of the paper.
10. A reviewer or a sub-reviewer can only submit a review once.
11. A PC member a can read a review for a paper p, provided the review has been submitted and a is not p's author and does not have a review outstanding for p.
12. A reviewer or a sub-reviewer may update the content of her review before she submits it.

To model this example in the RW formalism two classes need to be defined: `Agent` and `Paper`. The former is the set of agents. The latter is the set of papers. To model relations between these two sets, we need a number of predicates from which the set of propositional variables, P, is built. For $a, b \in$ `Agent`, $p \in$ `Paper`, P includes:

`author`(p, a)	a is an author of p,
`pcmember`(a)	a is a PC member,
`chair`(a)	a is the chair of the PC,
`reviewer`(p, a)	p is assigned to PC member a for reviewing,
`subreviewer`(p, a, b)	p is assigned by PC member a to sub-reviewer b,
`submittedreview`(p, a)	a review for p has been submitted by (sub-)reviewer a,
`review`(p, a)	the reviewing result for p from a, for simplicity represented as a boolean indicating 'accept' or 'reject'.

For each $p \in P$, $a \in \mathtt{Agent}$, the formulas $\mathtt{r}(p,a)$ and $\mathtt{w}(p,a)$ are defined using variables in P and logical connectors: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\exists$ (existential quantification) and $\forall$ (universal quantification), as follows. (For two agents a and b, $a = b$ denotes that a and b are the same agent, $\top$ denotes the boolean value $\mathtt{true}$, $\mathbf{\Sigma}$ denotes the set $\mathtt{Agents}$, and '$\rightleftharpoons$' denotes 'is defined as'.)

$$\begin{array}{ll}
\mathtt{r(author}(p,a),x) \rightleftharpoons \top & \\
\mathtt{r(pcmember}(a),x) \rightleftharpoons \top & \\
\mathtt{r(chair}(a),x) \rightleftharpoons \top & \\
\mathtt{w(author}(p,a),x) \rightleftharpoons \bot & \text{rule 1} \\
\mathtt{w(pcmember}(a),x) \rightleftharpoons \mathtt{chair}(x) \vee (\mathtt{pcmember}(x) \wedge a = x) & \text{rule 2}
\end{array}$$

$$\begin{array}{ll}
\mathtt{r(reviewer}(p,a),x) & \\
\quad \rightleftharpoons \mathtt{pcmember}(x) \wedge \neg\mathtt{author}(p,x) & \text{rule 4}
\end{array}$$

$$\begin{array}{ll}
\mathtt{w(reviewer}(p,a),x) & \\
\quad \rightleftharpoons \left(\begin{array}{l} (\mathtt{chair}(x) \wedge \mathtt{pcmember}(a) \wedge \neg\mathtt{author}(p,a)) \\ \vee \left(\begin{array}{l} \mathtt{pcmember}(x) \wedge x = a \wedge \mathtt{reviewer}(p,x) \\ \wedge \neg(\exists\, b \in \mathbf{\Sigma}\ \mathtt{subreviewer}(p,x,b)) \end{array} \right) \end{array} \right) & \text{rules 3, 6}
\end{array}$$

$$\begin{array}{ll}
\mathtt{r(subreviewer}(p,a,b),x) & \\
\quad \rightleftharpoons (\mathtt{pcmember}(x) \wedge \neg\mathtt{author}(p,x)) \vee x = b \vee x = a & \text{rule 7}
\end{array}$$

$$\begin{array}{ll}
\mathtt{w(subreviewer}(p,a,b),x) & \\
\quad \rightleftharpoons \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \mathtt{reviewer}(p,x)\ \wedge\ \neg\mathtt{author}(p,b) \\ \wedge\ x = a \end{array} \right) \\ \wedge\ \neg\exists\, d \in \mathbf{\Sigma} \left(\begin{array}{l} \mathtt{subreviewer}(p,a,d) \\ \vee\ \mathtt{subreviewer}(p,d,b) \end{array} \right) \end{array} \right) \\ \vee \left(\begin{array}{l} (\mathtt{subreviewer}(p,a,x) \wedge x = b) \\ \wedge\ \neg\mathtt{submittedreview}(p,x) \end{array} \right) \end{array} \right) & \text{rules 5, 8}
\end{array}$$

$$\begin{array}{ll}
\mathtt{r(submittedreview}(p,a),x) & \\
\quad \rightleftharpoons \mathtt{pcmember}(x) \wedge \neg\mathtt{author}(p,x) & \text{rule 9} \\
\mathtt{w(submittedreview}(p,a),x) & \\
\quad \rightleftharpoons \left(\begin{array}{l} \left(\begin{array}{l} (x = a) \wedge \\ \left(\begin{array}{l} \exists\, b \in \mathbf{\Sigma}\ \mathtt{subreviewer}(p,b,x) \\ \vee\ \mathtt{reviewer}(p,a) \end{array} \right) \end{array} \right) \\ \wedge\ \neg\mathtt{submittedreview}(p,x) \end{array} \right) & \text{rule 10}
\end{array}$$

$$
\begin{array}{ll}
\mathtt{r}(\mathtt{review}(p,a),x) & \\
\quad \rightleftharpoons \left(\begin{array}{l} \begin{pmatrix} \mathtt{pcmember}(x) \wedge \neg\mathtt{author}(p,x) \\ \wedge \mathtt{submittedreview}(p,a) \end{pmatrix} \\ \wedge \left(\begin{array}{l} \left(\begin{array}{l} \mathtt{reviewer}(p,x) \\ \rightarrow \mathtt{submittedreview}(p,x) \\ \wedge\, (\exists\, b \in \Sigma\ \mathtt{subreviewer}(p,b,x) \\ \rightarrow \mathtt{submittedreview}(p,x)) \end{array} \right) \\ \vee\ x = a \end{array} \right) \end{array} \right) & \text{rule 11}
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{w}(\mathtt{review}(p,a),x) & \\
\quad \rightleftharpoons \left(\begin{array}{l} (x = a \wedge \neg\mathtt{submittedreview}(p,x)) \wedge \\ \big(\exists\, b \in \Sigma\ \mathtt{subreviewer}(p,b,x) \vee \mathtt{reviewer}(p,x)\big) \end{array} \right) & \text{rule 12}
\end{array}
$$

In this example, what is not explicitly permitted is prohibited.

3.3. Discussions on the RW formalism

The RW formalism uses propositional variables to represent data, relations between data, and even permissions in an access control system. Rules in the access control policy adopted by the system are expressed by logical formulas built from the variables. The RW formalism is able to model a wide range of access control policies because of its abilities to express the following important concepts/features in access control systems.

Conditional authorisations. According to [12], this is the principle that protection requirements need to depend on the evaluation of conditions, normally including agents' roles, identities and so on. The ability of expressing conditional authorisation by logical formulas is a central feature of the RW formalism.

Permissions about permissions. Permissions about permissions are the permissions which can change other agents' permissions. In some literature, they are also called 'administrative policies' [27] or 'meta-policies' [7]. The RW formalism's ability to express permissions about permissions lies in the fact that it regards permissions as data along with other ordinary data in a system. For example, in the modelling of Example 3.1, $\mathtt{w}(\mathtt{reviewer}(p,a),x)$ defines the permission for changing $\mathtt{reviewer}(p,a)$, which, itself, expresses a's permission for reviewing p.

Delegation mechanisms. Delegation is the ability of one agent to give part or all its privileges to another so that the latter can carry out actions on behalf of the former [6]. Delegation may be used to help avoid the problem of *root bottleneck*, that is, the need for root or superuser to be involved in transfer of responsibilities. The RW formalism's ability to express delegation is demonstrated by the way that a reviewer can appoint sub-reviewers in Example 3.1.

Constraints. Constraints are important to most access control systems. In fact, sometimes, it is argued that constraints are the principal motivation behind the RBAC models [28]. Constraints can be expressed by the RW formalism because they can be modelled as relations between data of the system represented by the propositional variables and are integrated into the formulas representing authorisation rules.

3.4. The description language

The RW language is a machine-readable language designed for expressing access control policies modelled in the RW formalism and properties to be verified. A complete model written in the RW language consists of three parts: a program, a run-statement and a specification.

$$\langle \text{Model} \rangle ::= \langle \text{Program} \rangle [\langle \text{RunStatement} \rangle][\langle \text{Specification} \rangle]$$

The program describes the access rules. The run-statement specifies the size of the classes defined in the program. During the check, sets of those sizes will be created, thus defining a concrete instance on which the property will be evaluated. The specification is the property to be verified. In this subsection, we discuss issues about the syntax and the semantics of the program. We leave the discussion of the run-statement and the specification to Section 4.

3.4.1. Example 3.1 written in the RW language

Before formally discussing the syntax and the semantics of the description part, we shall see, in Fig. 1, the RW script for the policy defined in Example 3.1.

3.4.2. Syntax and semantics of the description part

A complete definition of the syntax can be found in Appendix A. Here, following the script in Fig. 1, we explain how a description is written in the RW language.

The program description starts with the keyword `AccessControlSystem`, followed by an identifier to name the model. Identifiers begin with a letter and can include letters, digits, "-" and "_". They are case-sensitive. The keyword `AccessControlSystem` and the identifier 'conference', form the *title* of this description.

What follows is the body of the description, which consists of a variable-definition and a policy-definition. The variable-definition defines classes and parameterised predicates. The policy-definition defines access rights of each agent on the variables defined in the variable-definition.

A class definition starts with the keyword `Class`, followed by any number of identifiers, which are interpreted as names of sets of objects. They must start with a capital letter. The class of agents is predefined and has the name `Agent`. That is why the example in Fig. 1 has only one class defined, which stands for the set of papers.

```
AccessControlSystem Conference
 Class Paper;
 Predicate author(paper: Paper, agent: Agent),
           pcmember(agent: Agent), chair(agent: Agent)!,
           reviewer(paper: Paper, agent: Agent),
           subreviewer(paper: Paper, appointer:Agent,
                       appointee: Agent),
           submittedreview(paper: Paper, agent: Agent),
           review(paper: Paper, agent: Agent);
 author(p, a){ read : true;
 }
 chair(a){ read: true;
 }
 pcmember(a){ read : true;
   write : (chair(user) | (pcmember(user) & a=user));
 }
 reviewer(p, a){
   read : pcmember(user) & ~author(p, user);
   write : (chair(user) & pcmember(a) & ~author(p,a))
             | ((pcmember(user) & user=a & reviewer(p,user))
                  & ~(E b: Agent [subreviewer(p,user,b)]));
 }
 subreviewer(p, a, b){
   read : (pcmember(user) & ~author(p,user)) | user=b
                                             | user=a;
   write : (reviewer(p,a) & ~author(p,b) & user=a
              & ~(E d: Agent [subreviewer(p,a,d) |
                              subreviewer(p,d,b)]))
             | (subreviewer(p,a,b) & ~submittedreview(p,b)
                & user=b);
 }
 submittedreview(p, a){
   read : pcmember(user) & ~author(p, user);
   write : (user=a) & ((E b: Agent [subreviewer(p, b, user)])
             | reviewer(p, user)) & ~submittedreview(p, user);
 }
 review(p, a){
   read : pcmember(user) & ~author(p, user)
            & submittedreview(p, a)
            & (((reviewer(p, user)
            -> submittedreview(p, user))
              and (E b: Agent [subreviewer(p, b, user)]
                    -> submittedreview(p, user))) | user=a);
   write : user=a & ((E b: Agent [subreviewer(p, b, user)])
                    | reviewer(p,a))
             & ~submittedreview(p, user);
 }
End
```

Fig. 1. The RW script for the policy defined in Example 3.1.

The predicate-definition starts with the keyword `Predicate`, followed by any number of predicates. Each predicate defines a logical relation. A predicate consists of a name and several parameters, whose types must be the defined classes. Parameter names must be distinct. Parameter names must start with lowercase letters. For example, the predicate-definition

$$\texttt{author(paper : Paper, agent : Agent)}$$

defines a parameterised predicate whose name is `author`. It has two parameters. The name of its first parameter is `paper`. Its type is the defined class `Paper`. The second parameter's name is `agent` and its type is the defined class `Agent`. By this definition, predicate `author` creates a relation between each element in the set `Paper` and each element in the set `Agent`. For each $p \in \texttt{Paper}$ and $a \in \texttt{Agent}$, $\texttt{author}(p, a)$ is a propositional variable which can be true or false, denoting whether a is an author of p or not. Any predicate marked by a '!' is a constant predicate which means among all the variables built from this predicate, only one of them is true and its value is unchangeable. Thus, `chair(agent : Agent)`! specifies that only one agent can be the chair of the PC.

The policy-definition consists of a number of rule-definitions. Each rule-definition begins with a parameterised predicate and should contain a pair of logical formulas labelled by the keywords `read` and `write`. These formulas define the conditions under which the agent, denoted by the keyword `user`, can *read* and *overwrite* the truth value of the variable represented by the instanced predicate. If one of the formulas or both formulas are omitted, the corresponding read or (and) write permission(s) is (are) also unavailable. Parameters in the brackets of the predicate can be used freely only within the block enclosed by the curly brackets. Thus each block creates a local name space. Variables defined in quantified formulas can only be used inside the quantified formulas. They are invisible from outside. For example, the following block (taken from Fig. 1)

```
pcmember(a){
  read : true;
  write : (chair(user) | (pcmember(a) and a=user));
}
```

defines that whether an agent a is a PC member is readable by all the agents, and is writable by the chair of the PC or by a himself. As the writing rule defines, the chair can set $\texttt{pcmember}(a)$ to be true, or false, meaning that the chair can both promote a to be a PC member and revoke a's membership. However, in the case of a himself, he can overwrite the truth value of $\texttt{pcmember}(a)$ only when he is already a PC member, meaning that he can preserve his membership or resign his membership. a cannot promote himself to be a PC member.

The logical connectives in a formula have their conventional meanings as they are used in Example 3.1. The program description ends at the keyword `End`.

4. Properties and their RW specifications

4.1. Run-statement

A system described by the program in the description part is only a template. To perform model-checking, a concrete instance based on the template needs to be constructed. This task is done through a run-statement. The syntax of the run-statement is:

⟨RunStatement⟩ ::= "run for" ⟨NumberClassPair⟩ ("," ⟨NumberClassPair⟩)*,
⟨NumberClassPair⟩ ::= ⟨Integer⟩ ⟨ClassName⟩.

When a run-statement is executed, the tool assigns each defined class a fixed number of elements. These elements are then used to instantiate the relations defined by the parameterised predicates. Once these two steps are finished, a concrete model with fixed size is established, on which model-checking is performed. Models of other sizes are not considered by the checking. Although large models may contain errors that small models cannot display, small models are still extremely useful for finding errors [23].

For the system defined by the program in Fig. 1, if the following line

run for 3 Paper, 4 Agent

is put after the keyword `End`, AcPeg will assign three elements to the set `Paper` and four elements to the set `Agent` when the statement gets executed. As a result, the set `Paper` becomes $\{p_1, p_2, p_3\}$, and the set `Agent` becomes $\{a_1, a_2, a_3, a_4\}$. Then, all the defined predicates are instantiated. For example, the predicate $\texttt{author}(\texttt{paper} : \texttt{Paper}, \texttt{agent} : \texttt{Agent})$ is instantiated to twelve propositional variables: from $\texttt{author}(p_1, a_1)$ to $\texttt{author}(p_3, a_4)$. After the population, the total number of the propositional variables in P is 104.

4.2. Strategies and guessing strategies

A *strategy* is a sequence of actions by which a coalition A of agents achieves a goal. Each action in the sequence is performed by an agent in A. We assume that an agent a performs actions only if he knows that he has permissions. (This is formalised in 5.6.1.)

A *guessing strategy* is similar to a strategy, except that it is not required that the agent knows he can perform each action. In this case, the agent is willing to take a risk that the action will be denied. The difference between a strategy and a guessing strategy is clarified using the following example.

EXAMPLE 4.1. Following Definition 3.1, we define an access control system $S\langle\Sigma, P, \texttt{r}, \texttt{w}\rangle$, where $P = \{u, x, y, z\}$ and $\Sigma = \{a\}$. For each $p \in P$, mappings $\texttt{r}(p, a)$ and $\texttt{w}(p, a)$ are summarised by Table 1.

Table 1

Mappings $\mathtt{r}(p, a)$ and $\mathtt{w}(p, a)$ for the access control system in Example 4.1

	u	x	y	z
$\mathtt{r}$	$\bot$	$\top$	$\top$	$\top$
$\mathtt{w}$	$\bot$	$\neg u$	u	$x \vee y$

Suppose the agent a has the goal to set z's value to false. There is no strategy to achieve this because a does not know the value of u, and therefore does not know whether to try setting x to false or to try setting y to false. But there is a guessing strategy, in which a 'guesses' the value of u and proceeds accordingly.

4.3. *Properties*

Following the run-statement, a property can be specified. A property is a query, taking the form of

$$\text{check } \{Ł \,\|\, \varphi \rightarrow A : \psi\},$$

where Ł defines a number of quantified variables used in φ, ψ and A; φ (optional) is a list of conditions based on which the goal, defined by ψ, is to be achieved; and A defines a coalition of agents who work together, intending to achieve the goal. What this query asks is: Are there *strategies* or *guessing strategies* – depending on the mode of checking – available for the agents in A, such that, by following the strategies or guessing strategies, they can achieve the goal defined by ψ on the basis of the conditions defined by φ.

A strategy is a sequence of *reading* and *overwriting* steps where in each step, there is an agent in A who knows she is permitted to take the action and so takes the action. What they read and overwrite are variables in P which stand for data, relations between data and permissions. A guessing strategy is similar to a strategy except that, following a guessing strategy, when an agent in A intends to read a variable, she does not need to have the proper permission. She can just read it. This reflects the possibility that intruders may acquire the information they need from other sources and thus in the course of reaching their goal, they may guess out the value of the data they need. AcPeg performs both modes of checking, where, in one mode, it searches for strategies, and, in the other, for guessing strategies.

In what follows, we shall explain the nature of Ł, φ, ψ, A separately, and finally we shall see a few example queries.

4.3.1. *Quantified variable definition*

Quantified variables in Ł are defined on the classes defined in the class-definition. They can be existential or universal, declared using 'E' and 'A' respectively. Quantified variables defined in the same class may represent the same element during the course of a checking, unless the keyword `disj` is used.

4.3.2. Conditions

Conditions defined in φ serve as pre-requirements on which the checking is based. Each condition is a variable in the set P. It can be positive or negative. Conditions in φ are connected by logical conjunction ($\wedge$). For a $p \in P$ which occurs in φ, we summarise its possible forms and their meanings in the following list. ($\top$ stands for the boolean value `true` and $\perp$ stands for `false`.)

p^*	p's value is constant during the checking. However, this value is unknown by the agents in A,
$(\neg)p!$	p is $\top(\perp)$ at the beginning of the checking and there is at least an agent in A who knows p is $\top(\perp)$ at the beginning of the checking,
$(\neg)p{*}!$	p is $\top(\perp)$ constantly during the course of the checking and there is at least an agent in A who knows p is $\top(\perp)$ at the beginning of the checking.

The forms $(\neg)p!$ and $(\neg)p{*}!$ qualify the agents' knowledge states. If a predicate is marked by a '!' (constant predicate) in the predicate-definition, the user should explicitly mark the supposed constantly true variable by '*!' in the condition (see the condition in Query 4.2). Otherwise the checker does not know which variable is constantly true and thus ignores this constraint of cardinality.

Note that the exclamation mark ('!') is used in two ways in our notation. In the predicate definition (see 3.4.2) it is used to mark a constant predicate. In the conditions described in this section, it is used to specify that the value of a variable is known by the agents in A.

4.3.3. Goals

A goal defined by ψ can be a *simple goal* or a *nested goal*. A simple goal is a combination consisting of conjunction and disjunction of three kinds of atomic goals. These are *making* goals, *realising* goals and *reading* goals, written using '{ }', '$\langle\ \rangle$' and '[ ]', respectively. If l_1, l_2 and l_3 are propositional formulas belonging to the set $\mathbf{L}(P)$ defined in Definition 3.1, $\{l_1\}$ is the goal of *making* l_1 true; $\langle l_1\rangle$ is the goal of *realising* that l_1 is true; and $[l_1]$ is the goal of finding out the truth value of l_1, whatever this value is. 'Making' goals mean changing the system state to bring about certain conditions. 'Reading' goals are to extract information about the system state. 'Realising' goals are auxiliary and are used to allow the construction of conditionals such as $\langle l_1\rangle$ `and` $\{l_2\}$ `or` $\langle$`not` $l_1\rangle$ `and` $\{l_3\}$, which means: achieve either l_2 or l_3 according to whether l_1 is true or false. A single 'realising' goal $\langle l_1\rangle$ is unlikely to be useful, because l_1 may simply turn out to be false.

A nested goal is a goal which is composed by subgoals. The depth of nesting is unlimited. Generally, it has the form

$$\text{check}\ \big\{\text{Ł} \parallel \varphi \rightarrow\ A_1 : \big(\psi_1\ \texttt{AND}\ A_2 : (\psi_2\ \ldots\ \texttt{AND}\ A_n : (\psi_n)\ldots)\big)\big\},$$

where $\psi_1, \psi_2, \ldots, \psi_n$ are simple goals. Its meaning is: Are there strategies or guessing strategies available for agents in $A_1, A_2, \ldots$, and A_n, such that, if conditions in

φ are true, the agents in A_1 can achieve a state in which the goal ψ_1 is satisfied, and (in that state) the agents in A_2 can achieve the goal $\psi_2, \ldots$, and finally the agents in A_n can achieve the goal ψ_n. What this nested goal describes is a sequencing of actions performed by the agents in $A_1, A_2, \ldots$, and A_n. However, this sequencing is achieved only if the conditions defined in φ do not enable agents in A_i make other subgoals true while on the way of achieving ψ_i.

4.3.4. Coalition of agents

In the case of a simple goal, A defines the coalition who intends to achieve the goal ψ. In the case of a nested goal, each A_i defines the coalition who is to achieve ψ_i.

4.3.5. Example queries written in the RW language

In what follows we shall see a few queries (together with run-statements) written in the RW language for the system defined by the program in Figure 1.

QUERY 4.1.

```
run for 3 Paper, 4 Agent
check {E a: Agent, p: Paper || ~chair(a)*!
       -> {a}:{reviewer(p,a)}}
```

The query asks: Are there strategies or guessing strategies available for $a \in$ `Agent` such that, on knowing the fact that he is not the chair of the PC, a can promote himself to be a reviewer of a paper p. Curly brackets are used to enclose a to denote the set of acting agents.

QUERY 4.2.

```
run for 3 Paper, 4 Agent
check {E disj a,c: Agent, p: Paper || chair(c)*!
       -> {c}:{reviewer(p,a)}}
```

The query asks: Are there strategies or guessing strategies available for $c \in$ `Agent` such that, on knowing the fact that she is the chair of the PC, c can promote another agent a to be a reviewer of a paper p.

QUERY 4.3.

```
run for 1 Paper, 3 Agent
check {E disj a,b,c: Agent, p: Paper || chair(c)*! &
 ~submittedreview(p,a)! & submittedreview(p,b)*! &
 ~author(p,a)*! & pcmember(a)*! & ~reviewer(p,a)! &
 ~subreviewer(p,b,a)*! & ~subreviewer(p,c,a)*! &
 ~subreviewer(p,a,a)*!
 -> {a}:([review(p,b)] AND
 {a,c}:({submittedreview(p,a)}))}
```

This query is a one-level nested query which essentially asks for the strategy introduced in Strategy 1.1. The conditions are used deliberately to create a situation where such a strategy can be found. In particular, the conditions

```
~subreviewer(p,b,a)*! & ~subreviewer(p,c,a)*!
 & ~subreviewer(p,a,a)*!
```

are used because rule 11 defined in Example 3.1 requires that to read a paper's review, a PC member must not have an outstanding review for the paper.

QUERY 4.4.

```
run for 1 Paper, 3 Agent
check {E disj a,c :Agent || chair(c)*! and ~chair(a)*!
and ~pcmember(a)!
-> {c}:({pcmember(a)} AND {a}:({~pcmember(a)} AND
{c}:({pcmember(a)}
AND {a}:({~pcmember(a)} AND {c}:({pcmember(a)})))))}
```

This is a five-level nested query which asks: Are there strategies or guessing strategies available for $a, c \in$ `Agent` such that on knowing that c is the chair of the PC, a is not the chair and initially a is not a PC member, c can promote a to be a PC member, then a can resign his membership, then c can promote a again, and then a can resign and finally c can promote a once more.

5. RW model-checking

5.1. The RW model-checking problem

Given a query Q, such as the one defined in 4.3, the task of the RW model-checking algorithm is to figure out whether the strategies or guessing strategies queried by Q exist, and if they exist, output at least one of them.

5.2. Assumptions made by the algorithm

Given an access control system $S\langle\Sigma, P, \mathtt{r}, \mathtt{w}\rangle$, a query Q, which includes the conditions φ, the goal ψ and the group of agents A, the algorithm finds a strategy by modelling the accumulation of A's knowledge about the state of S while A carries out the strategy. Each step of the strategy is either reading the value of a $p \in P$ or overwriting the value of a $p \in P$ by an agent in A. The assumptions made by the algorithm in modelling the accumulation of A's knowledge while the carrying out of a non-guessing strategy are:

1. While carrying out the strategy, the agents are assumed to completely know the policy of S. A common principle in secure system design is to avoid relying on 'security by obscurity'. Thus, we must assume that the attacker knows any information about the design of the system that he could know.
2. At the beginning of executing a strategy, A holds no knowledge about the initial state of S, except for the values of the variables marked by '!' in φ.
3. While carrying out the strategy, at each step only an agent in A acts.
4. For a $p \in P$ and an $a \in A$, while carrying out the strategy, no matter a overwrites p's value or samples p's value, a does so only if a knows that he is permitted to do so. In other words, a overwrites p's value only if $\mathtt{w}(p, a)$ is true and a knows $\mathtt{w}(p, a)$ is true. It is the same when a samples a variable in P. The reason we made this assumption is that, we model the fact that a performs actions only if he knows he can. A strategy must guarantee the achievability of ψ, therefore, in each of its step, the agents must know the action of the step can be performed.
5. While carrying out the strategy, if an agent a in the set A overwrites or samples a variable p, a knows the resulting value of p. If a knows the value of p, the coalition A also knows it, because we assume agents in A can communicate with each other outside of the model.
6. While carrying out the strategy, an agent a samples a variable p only if p's value is not known by the coalition A. In other words, a samples p only if p has not been sampled or overwritten before. There is no point of sampling it if its value is already known.

In searching for a guessing strategy, the algorithm assumes agents in A can sample every variable even if they are not permitted to do so by the policy defined in S.

5.3. The transition system

The algorithm is built around the knowledge of the state of the system S that the considered coalition A has at each step of implementing its strategy. Obviously there is a set of knowledge states each of which is sufficient for A to regard its goal as achieved. This is so when A knows that the formulas in some appropriate combination of the involved making goals are true, enough is known to work out the truth values of the formulas in the reading goals, etc. We denote the set of the knowledge states from which A can deduce that its goal is achieved by K_{G}. Each step of a strategy takes A from a knowledge state to a possibly richer one until a state in K_{G} is reached. A knowledge state combines knowledge of the initial state of the system, that is, the state of the system at the beginning of executing a strategy, and knowledge of its current state. Assignments contribute the knowledge of the current value of the assigned variable, which has been just given to it. This means that learning and changing the system are done simultaneously. Sampling steps contribute A's knowledge on both the current and the initial value of the sampled variable.

Overwriting without sampling in advance destroys the prospect to learn the initial value of the variable. Strategies are supposed to take A from its initial knowledge state $k_{\mathtt{init}}$ to one in K_{G} from which its goal is deemed as achieved.

To describe A's knowledge on p, we use four knowledge variables. For each $p \in P$, we have

v_{0p}	is true if A knows the initial value of p,
t_{0p}	is true if A knows initially p is true,
v_p	is true if A knows the current value of p,
t_p	is true if A knows currently p is true.

When overwriting p to true, v_p and t_p both become true, but v_{0p} and t_{0p} do not change, because overwriting does not contribute A's knowledge on p's initial value. When overwriting p to false, v_p becomes true; t_p becomes false; both v_{0p} and t_{0p} do not change. When sampling p, v_{0p} and v_p both become true and t_{0p} and t_p both become false if p turns out to be false, or t_{0p} and t_p both become true if p turns out to be true. Since the contents of t_{0p} and t_p are irrelevant when p is unknown, and the initial value of a variable is known only if the current value is known too, there are indeed only 7, and not 2^4 knowledge states about each variable p. However it is easier to explain our algorithm in terms of v_{0p}, t_{0p}, v_p and t_p as independent variables.

A knowledge state is given by the quadruple (V_0, T_0, V, T), where

$$V_0 = \{p \in P \mid v_{0p} \ \mathtt{is} \ \top\}, \quad T_0 = \{p \in P \mid t_{0p} \ \mathtt{is} \ \top\},$$
$$V = \{p \in P \mid v_p \ \mathtt{is} \ \top\}, \qquad T = \{p \in P \mid t_p \ \mathtt{is} \ \top\}.$$

We show the effects that the above three kinds of transitions have on the knowledge states when a variable p is overwritten and sampled by an agent in Fig. 2.

Therefore, by modelling the accumulation of A's knowledge, we build a transition system over the access control system in question. Three kinds of transitional relations can be identified – *overwriting-to-true*, *overwriting-to-false* and *sampling*, each of which carries the knowledge states of A from one to another until A has confidence to deduce the goal is reached from its current knowledge state. Once A reaches a knowledge state from which it can deduce the goal is reached, we consider that its goal has been reached. Fig. 3 illustrates the above process, however in a simplified situation where only one variable p is considered.

Note the transition relation for overwriting are deterministic; the relation for sampling is not. A strategy should lead A to the goal through both possible outcomes of a sampling.

To find such a strategy which leads A from $k_{\mathtt{init}}$ to K_{G} the algorithm works backwards by inverting the process described above. Before discussing the backwards reachability computation, we shall first see how $k_{\mathtt{init}}$ and K_{G} are represented by

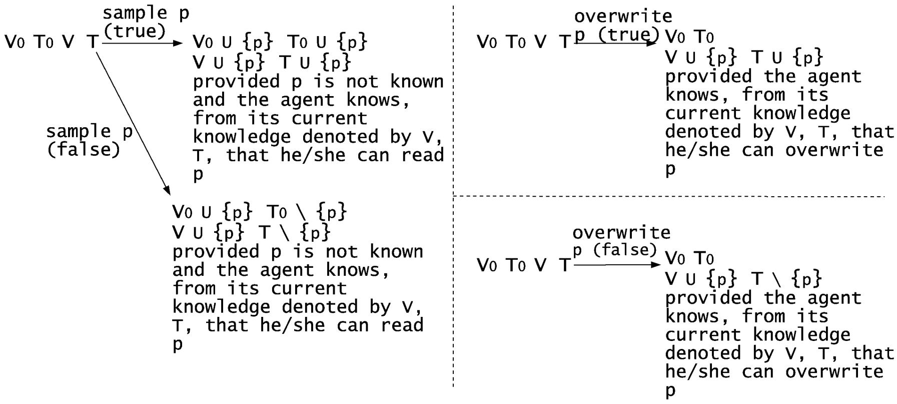

sample p
(true)
Vo To V T
Vo U {p} To U {p}
V U {p} T U {p}
provided p is not known and the agent knows, from its current knowledge denoted by V, T, that he/she can read p
sample p
(false)
Vo U {p} To \ {p}
V U {p} T \ {p}
provided p is not known and the agent knows, from its current knowledge denoted by V, T, that he/she can read p
overwrite
p (true)
Vo To V T
Vo To
V U {p} T U {p}
provided the agent knows, from its current knowledge denoted by V, T, that he/she can overwrite p
overwrite
p (false)
Vo To V T
Vo To
V U {p} T \ {p}
provided the agent knows, from its current knowledge denoted by V, T, that he/she can overwrite p

Fig. 2. The transitions.

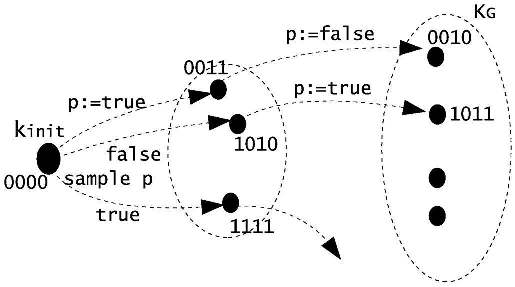


Fig. 3. The process of learning.

boolean expressions composed of the knowledge variables. Using boolean expressions, which then are represented by BDDs, to represent sets of states is the typical approach used in symbolic model-checking.

5.4. *Representation of* $k_{\mathtt{init}}$

According to our assumptions in Section 5.2, $k_{\mathtt{init}}$ is the state where A knows nothing about the state of S, except on the values of the variables marked by '!' in the conditions defined by φ. This means that for all variables in P which also occur in φ and are marked by '!', A knows their values. However, for all the other variables in P, A does not know their values initially. Now for each variable $p \in P$, we have four knowledge variables v_{0p}, t_{0p}, v_p and t_p to describe A's initial and current knowledge about it. We use a boolean expression composed of the knowledge variables to represent $k_{\mathtt{init}}$. This boolean expression is then represented by a BDD in the course of the RW model-checking.

We divide variables in P into three mutually-exclusive subsets. P^+ is the set for variables in P which only occur positively in φ and are marked by '!' and by '*!'.

P^- is the set for variables in P which only occur negatively in φ and are marked by '!' and by '*!'. P° is the set for all the other variables in P. Now for each $p \in P^+$, we use $(v_{0p} \wedge t_{0p} \wedge v_p \wedge t_p)$ to represent A's initial knowledge about p. In the case that $p \in P^-$, we use $(v_{0p} \wedge \neg t_{0p} \wedge v_p \wedge \neg t_p)$ to represent A's initial knowledge about it; and finally, if $p \in P^\circ$, we use $(\neg v_{0p} \wedge \neg t_{0p} \wedge \neg v_p \wedge \neg t_p)$ to represent A's initial knowledge about it. Therefore, the representation of $k_{\mathtt{init}}$ by the knowledge variables is the conjunction of all the representations of the above forms for variables in P.

5.5. *Representation of K_{G}*

Given a formula ψ describing the goal we want to produce a representation of the goal states. ψ is a conjunction and disjunction combination of reading goals $[l]$, realising goals $\langle l \rangle$, and making goals $\{l\}$, where l in each case is a boolean combination of variables of P.

We want to represent ψ as a set of knowledge states, being those in which the goal is known to be true. A set of knowledge states can be represented by a formula over the propositions $\{v_{0p}, t_{0p}, v_p, t_p \mid p \in P\}$. To relate this to the notation of Fig. 2, note that v_{0p} is true in a state if $p \in V_0$, t_{0p} is true if $p \in T_0$, and similarly for v_p and t_p.

Suppose an agent's knowledge of the state of the system is represented by V, T. Then a formula over $\{v_{0p}, t_{0p}, v_p, t_p \mid p \in P\}$ expressing the agents ability to determine that l is true may be constructed as follows:

- if v_p is true, then substitute t_p for p in l. This covers the case that the agent knows the value of p;
- if v_p is false, then replace l with a version in which $\top$ is substituted for p and another in which $\perp$ is substituted for p. This covers the case that the agent does not know p.

Thus, the formula expressing the agent's ability to determine that l is true is

$$(l[t_p/p] \wedge v_p) \vee (l[\top/p] \wedge l[\perp/p] \wedge \neg v_p).$$

In the following definition, we generalise this formula to consider the effect of all the $p \in P$.

Definition 5.1. Let V, T be the knowledge held by an agent, and l a formula over the propositions P. The propositions v_p and t_p signify $p \in V$ and $p \in T$, respectively. The formula

$$\gamma_{V,T} l = \left(\bigwedge_{S \subseteq p} l[(v_p \rightarrow p)/p \mid p \in S][(v_p \wedge p)/p \mid p \in P \backslash S] \right) [t_p/p \mid p \in P]$$

expresses the agent's ability to determine that l is true. The $S \subseteq P$ produces all the possible combinations of $\top$ and $\perp$ to substitute ps such that v_p is false. Note that

$v_p \rightarrow p$ is $\top$ when v_p is false, and p otherwise; similarly, $v_p \wedge p$ is $\bot$ if v_p is false and p otherwise. Hence, S just enumerates all the vectors of $\top$s and $\bot$s for the ps and $v_p \rightarrow p$ and $v_p \wedge p$ are used to restrict the effect of the substitution only to ps such that v_p is false.

5.5.1. *Substitution of reading goals*

The knowledge states in which the reading goal $[l]$ is known to be achieved are those in which the knowledge held is sufficient to evaluate l in V_0, T_0. In order to do that, the agent needs to be able to determine that l is true, or that it is false. Thus, the appropriate formula over $\{v_{0p}, t_{0p}, v_p, t_p \mid p \in P\}$ is $\gamma_{V_0,T_0} l \vee \gamma_{V_0,T_0}(\neg l)$.

5.5.2. *Substitution of realising goals*

The knowledge states in which the realising goal $\langle l \rangle$ is known to be achieved are those in which the knowledge held is sufficient to evaluate l in V_0, T_0. To realise that l is true, l has to be true. Thus, the appropriate formula over $\{v_{0p}, t_{0p}, v_p, t_p \mid p \in P\}$ is $\gamma_{V_0,T_0} l$.

5.5.3. *Substitution of making goals*

The knowledge states in which the making goal $\{l\}$ is known to be achieved are those in which the knowledge held is sufficient to evaluate that l is true in V, T. Thus, the appropriate formula over $\{v_{0p}, t_{0p}, v_p, t_p \mid p \in P\}$ is $\gamma_{V,T} l$.

5.6. *Backwards reachability computation*

5.6.1. *Computing sets of states*

To find strategies the algorithm starts from K_{G}, searching for sets of the knowledge states which transition into K_{G} by overwriting any $p \in P$ to true; sets of the knowledge states which transition into K_{G} by overwriting any $p \in P$ to false; and sets of the knowledge states which transition into K_{G} by sampling any $p \in P$. Then for each newly found set, the algorithm continues to find other sets of the knowledge states which transition into the new set through either of the three kinds of transition relations. During this process, if $k_{\mathtt{init}}$ is found in a set of knowledge states, the goal is considered as reachable by following the operations represented by the transition relations which connect the set in which $k_{\mathtt{init}}$ is found to K_{G}. The operations along the path are deemed as the steps of a strategy by the algorithm.

In order to formally describe this process, we shall first define the concept of presets. For any $a \in A$, $p \in P$ and a given set of knowledge states Y,

$\mathtt{Pre}^{\exists,a}_{p:=\top}(Y)$ is the set of the knowledge states in which a knows she is permitted to overwrite p and which transition into Y by overwriting p to true ($\top$). Its formal definition is: $\{(V_0, T_0, V, T) \mid (V_0, T_0, V \cup \{p\}, T \cup \{p\}) \in Y, \gamma_{V,T}\mathtt{w}(p, a) = \top\}$. The conditions in this set comprehension are conjoined together.

$\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y)$ is the set of the knowledge states in which a knows she is permitted to overwrite p and which transition into Y by overwriting p to false ($\bot$). Its formal definition is: $\{(V_0, T_0, V, T) \mid (V_0, T_0, V \cup \{p\}, T \setminus \{p\}) \in Y, \gamma_{V,T}\mathtt{w}(p, a) = \top\}$.

$\mathtt{Pre}^{\exists,a}_{p=\top}(Y)$ is the set of the knowledge states in which a knows he is permitted to sample p and which transition into Y by sampling p and find out it is true ($\top$). Its formal definition is: $\{(V_0, T_0, V, T) \mid p \notin V_0, p \notin T_0, p \notin V, p \notin T, (V_0 \cup \{p\}, T_0 \cup \{p\}, V \cup \{p\}, T \cup \{p\}) \in Y, \gamma_{V,T}\mathtt{r}(p, a) = \top\}$.

$\mathtt{Pre}^{\exists,a}_{p=\bot}(Y)$ is the set of the knowledge states in which a knows he is permitted to sample p and which transition into Y by sampling p and find out it is false ($\bot$). Its formal definition is: $\{(V_0, T_0, V, T) \mid p \notin V_0, p \notin T_0, p \notin V, p \notin T, (V_0 \cup \{p\}, T_0 \setminus \{p\}, V \cup \{p\}, T \setminus \{p\}) \in Y, \gamma_{V,T}\mathtt{r}(p, a) = \top\}$.

In the above definitions, $\gamma_{V,T}\mathtt{r}/\mathtt{w}(p, a) = \top$ denotes the condition under which a knows with Knowledge V, T that she is permitted to read/overwrite p. The mapping $\mathtt{r}/\mathtt{w}(p, a)$ is a boolean expression composed of variables in P. It defines the condition under which a is permitted to read/overwrite p. To represent a's current knowledge on $\mathtt{r}/\mathtt{w}(p, a)$, we need to use the knowledge variables in V and T to replace every occurrence of variables in $\mathtt{r}/\mathtt{w}(p, a)$, because variables in V and T describes A's knowledge (a and A share the same knowledge) about the current state of the system. The principle for the substitution is essentially the same with the principle we use in representing K_{G}.

In Section 5.5 we have seen how K_{G} is represented by a boolean expression. The algorithm uses BDDs to represent boolean expressions. All the pre-sets can be obtained from K_{G} through BDD computations using the formula in [21]:

$$\mathtt{Pre}^{\exists}_{\rightarrow}(Y) = \mathtt{exists}\big(\hat{y}', \mathtt{apply}(\mathtt{and}, B_{\rightarrow}, B_{Y'})\big),$$

where $B_{Y'}$ is the BDD representation for the set Y' (which is obtained by replacing all the variables in Y by their primed versions), $B_{\rightarrow}$ is the BDD representation for the transition relation $\rightarrow$, and $\hat{y}'$ stands for all the primed variables. In our case, $B_{\rightarrow}$ is obtained by synthesising all the conditions in the formal definition of $\mathtt{Pre}^{\exists,a}_{\rightarrow}$ discussed above.

5.6.2. *Generating strategies*

During the course of the computation, the algorithm maintains pairs (Y, s) consisting of a set Y of knowledge states and a strategy s. The pair (Y, s) denotes the fact that s is a strategy that enables A to reach K_{G} from each state in Y. For K_{G}, the s is simply '`skip;`', which means 'do nothing'.

The algorithm starts with the pair (K_{G}, `skip;`). The core of the algorithm works as follows: Given the pair (Y, s), it adds the pairs $(\mathtt{Pre}^{\exists,a}_{p:=\top}(Y), (p := \top; s))$ and $(\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y), (p := \bot; s))$. For any two pairs (Y_1, s_1) and (Y_2, s_2), it adds the pair $(\mathtt{Pre}^{\exists,a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{p=\bot}(Y_2)$, **if** (p) by a **then** s_1 **else** $s_2)$.

The algorithm continues until no new pairs are generated. Now, all the pairs whose set of knowledge states contains $k_{\mathtt{init}}$ contain the strategies we are looking for.

To find out guessing strategies instead of strategies, the only thing that needs to be changed is to omit the condition $\gamma_{V,T}\mathtt{r}(p,a) = \top$ when computing $\mathtt{Pre}^{\exists,a}_{p=\top}(Y)$ and $\mathtt{Pre}^{\exists,a}_{p=\bot}(Y)$.

5.7. Pseudo-code for the algorithm

The algorithm for extracting strategies is described below in the form of pseudo-code. It assumes as input the initial state $k_{\mathtt{init}}$ and the set of goal knowledge states $K_{\mathtt{G}}$. It outputs at least a strategy for going from $k_{\mathtt{init}}$ to some state in $K_{\mathtt{G}}$. The algorithm works by backwards reachability from $K_{\mathtt{G}}$ to $k_{\mathtt{init}}$. It maintains a set of states it has seen, called `states_seen`, and a data structure storing the pairs found so far, called `strategies`.

Input: $K_{\mathtt{G}}$ – set of goal knowledge states, $k_{\mathtt{init}}$ – the initial knowledge state,
P – set of propositional variables, A – set of acting agents,
`r`, `w` – mappings defining reading and writing permissions (are used when computing the pre-sets, though not explicitly shown in the algorithm).
Output: a strategy for going from $k_{\mathtt{init}}$ to some state in $K_{\mathtt{G}}$ if such strategies exist.

```
strategies := $\emptyset$;
states_seen := $\emptyset$;
put ($K_{\mathtt{G}}$, skip;) in strategies;
repeat until strategies does not change{
    choose $(Y_1, s_1) \in$ strategies;   // for all pairs in strategies
    for each $p \in P${
        for each $a \in A${
            $PTY_1$ := $\mathtt{Pre}^{\exists,a}_{p:=\top}(Y_1)$;
            if (($PTY_1 \neq \emptyset$) $\wedge$ ($PTY_1 \nsubseteq$ states_seen)){
                states_seen := states_seen $\cup$ $PTY_1$;
                $pts_1$ := "set $p$ to $\top$ by $a$;" + $s_1$;
                strategies := strategies $\cup$ $(PTY_1, pts_1)$;
                if ($k_{\mathtt{init}} \in PTY_1$)
                    output $pts_1$;
            }
            $PFY_1$ := $\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y_1)$;
            if (($PFY_1 \neq \emptyset$) $\wedge$ ($PFY_1 \nsubseteq$ states_seen)){
                states_seen := states_seen $\cup$ $PFY_1$;
                $pfs_1$ := "set $p$ to $\bot$ by $a$;" + $s_1$;
                strategies := strategies $\cup$ $(PFY_1, pfs_1)$;
                if ($k_{\mathtt{init}} \in PFY_1$)
                    output $pfs_1$;
            }
        }
    }
    choose $(Y_2, s_2) \in$ strategies;    // for all pairs in strategies
    for each $p \in P${
        for each $a \in A${
            $PSY$ := $\mathtt{Pre}^{\exists,a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{p=\bot}(Y_2)$;
            if (($PSY \neq \emptyset$) $\wedge$ ($PSY \not\subseteq$ states_seen)){
                states_seen := states_seen $\cup$ $PSY$;
                strategies := strategies $\cup$ ($PSY$, $pss$);
                $pss$ := "if ($p$) by a then $s1$ else $s2$";
                if ($k_{\mathtt{init}} \in PSY$)
                    output $pss$;
            }
        }
    }
}
```

In practice, we found there is another way to compute the pairs, which is only slightly different from the one described above. We provide its pseudo-code in Appendix B, where we use bold font to highlight the differences. We call the algorithm described here 'Algo-0' and the one described in Appendix B 'Algo-1'. When there are no strategies, both Algo-0 and Algo-1 find none. Because in these cases, $k_{\mathtt{init}}$ will not be found in any set generated during the pre-computations. When there are some strategies, both Algo-0 and Algo-1 find some, however, the strategies found by Algo-1 may differ from the ones found by Algo-0. AcPeg integrates both Algo-0 and Algo-1, so that the user can use either of them. However, because Algo-0 is easier to reason about than Algo-1, we present Algo-0 here and use it as the basis for our analysis.

5.8. *Proof of correctness*

THEOREM 1. *The algorithm will eventually terminate.*

PROOF. To prove the algorithm will terminate is equivalent to proving that the size of `strategies` will not infinitely grow. The set `strategies` only increases if we encounter states not yet in `states_seen`. As there are only finitely many states, we cannot go on encountering new states for ever. □

LEMMA 1. *If there exists a strategy s, then the algorithm will produce a strategy (but not necessarily s).*

PROOF. The algorithm performs exhaustive backwards reachability and therefore will find all states for which there is a strategy to arrive at K_{G}.

Since s is a strategy from $k_{\mathtt{init}}$ to K_{G}, $k_{\mathtt{init}}$ will be in the set of states found and therefore the algorithm will output a strategy from $k_{\mathtt{init}}$ to K_{G}. □

REMARK 1. The algorithm is non-deterministic thanks to the choice operator. That is why it is not guaranteed to obtain s in the lemma above. The algorithm prevents any subset from being added to `strategies` if all the states in that subset have already been found in `states_seen`. This condition guarantees the termination of the algorithm, however, at the cost of eliminating the prospect of exploring some strategies. In all cases there is a way of picking the choices so that s is output.

THEOREM 2. *If there are strategies from* $k_{\mathtt{init}}$ *to* K_{G} *the algorithm finds at least one of them.*

PROOF. Following Lemma 1, however the choice operator is resolved, $k_{\mathtt{init}}$ will eventually be included in `states_seen`, and therefore some strategy will be generated. □

LEMMA 2. *For all* $(Y, s) \in$ `strategies`*, and for all* $y \in Y$*,* s *succeeds on* y *and the result is in* K_{G}*.*

PROOF. We look at all the ways that (Y, s) can be added to `strategies`. At the beginning, (K_{G}, `skip`;) is added in. The correctness of the lemma is self-evident for this case. During the course of the algorithm, pairs are added in one of these three circumstances:

(i) (PTY_1, pts_1) is added, where, $\exists\ a \in A$ and $p \in P$, such that $PTY_1 = \mathtt{Pre}^{\exists,a}_{p:=\top}(Y_1)$, $pts_1 =$ "set p to $\top$ by a;" + s_1, and (Y_1, s_1) is in `strategies`.
We know by the inductive hypothesis for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in K_{G}. We also know for all $y \in PTY_1$ that a can do $p := \top$ and that the result of that is in Y_1, because that is the way we get PTY_1 from Y_1. Therefore pts_1 succeeds on all the states in PTY_1 and the result is in K_{G}.

(ii) (PFY_1, pfs_1) is added, where, $\exists\ a \in A$ and $p \in P$, such that $PFY_1 = \mathtt{Pre}^{\exists,a}_{p:=\bot}(Y_1)$, $pfs_1 =$ "set p to $\bot$ by a;" + s_1, and (Y_1, s_1) is in `strategies`.
The argument for the above case applies also to this one.

(iii) (PSY, pss) is added, where, $\exists\ a \in A$ and $p \in P$, such that $PSY = \mathtt{Pre}^{\exists,a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{p=\bot}(Y_2)$, $pss =$ "if (p) by a then $s1$ else $s2$" and (Y_1, s_1), and (Y_2, s_2) are both in `strategies`.
We know by the inductive hypothesis for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in K_{G}, and $y_2 \in Y_2$, s_2 succeeds on y_2 and result is in K_{G}. We also

know for all $y \in PSY$ that a can read p and if it is $\top$, the result of that is in Y_1. However, if it is $\bot$, the result of that is in Y_2. Therefore pss succeeds on all the states in PSY and the result is in K_G. □

THEOREM 3. *If the algorithm outputs the strategy s then s succeeds on $k_\mathtt{init}$ and the result is in K_G.*

PROOF. From Lemma 2 we know that for all $(Y, s) \in \mathtt{strategies}$ and $y \in Y$, s succeeds on y and the result is in K_G. Because if s gets output, there must exist a Y, such that $k_\mathtt{init} \in Y$ and $(Y, s) \in \mathtt{strategies}$. Therefore, it follows that s succeeds on $k_\mathtt{init}$ and the result is in K_G. □

From the implication of theorem 3, we know that if there is no strategy s which succeeds on $k_\mathtt{init}$ and results in K_G, the algorithm will output none.

5.9. Computational complexity

We use K for the set of all the knowledge states, $|K|$ for the total number of knowledge states, $|P|$ for the number of variables in P, $|A|$ for the number of acting agents. We assume that $|K|$ is equal to $2^{4|P|}$ because for each variable in P, we use four knowledge variables to represent the coalition's knowledge about it. (This is an upper bound; $|K|$ is actually less than $2^{4|P|}$ because not all the knowledge variables are independent.)

Because the computations in our algorithm are done through operations between BDDs, several remarks concerning the complexity of BDD operations should be made in advance.

- For two BDDs B_1 and B_2, the complexity of the operation $\mathtt{apply(and/or,}$ $B_1, B_2\mathtt{)}$ is determined by $|B_1||B_2|$, where $|B_1|$ and $|B_2|$ are the numbers of nodes in B_1 and B_2 respectively.
- Suppose X and Y are two subsets of K; B_X and B_Y are the BDD representations of X and Y respectively. The number of nodes in B_X or B_Y is at most $2^{4|P|}$. Therefore, the complexity of the operation $\mathtt{apply(and/or,}B_X, B_Y\mathtt{)}$ in the worst case is $2^{8|P|}$.
- Suppose $B_\rightarrow$ is a BDD representing one of the transition relations presented in Section 5.6.1, the number of nodes in $B_\rightarrow$ is at most $16|P|$ if the knowledge variables are properly ordered.
- The complexity of an existential quantification over n variables in a BDD is 2^n.
- Checking the equality of two BDDs takes constant time in all BDD implementations, e.g., BuDDy.

In what follows, we discuss the complexity of the algorithm line by line, only omitting those lines of which operations spend constant time.

Line 5. Because the conditions in Lines 9, 17 and 30 prevent any subset whose elements are already found from being added to `strategies`, and, in the worst case, the subsets of K are just singletons, there are at most $|K|$ ($2^{4|P|}$) pairs in `strategies`. Therefore, this step repeats at most $2^{4|P|}$ times.

Line 6. This step repeats $|P|$ times.

Line 7. This step repeats $|A|$ times.

Line 8. According to the formula for computing $\mathtt{Pre}^{\exists}_{\rightarrow}(Y)$ in Section 5.6.1, the operation of this step involves the BDD computation $\mathtt{apply}(\mathtt{and}, B_{p:=\top}, B_{Y'_1})$ followed by an existential quantification over all the V', T' variables on the resulting BDD of the `apply` operation. Let $|B_{p:=\top}|$ denote the number of nodes in $B_{p:=\top}$ and $|B_{Y'_1}|$ denote the number of nodes in $B_{Y'_1}$. $2|P|$ is the total number of V', T' variables. The complexity of the `apply` operation in the worst case is $16|P|2^{4|P|}$ and the complexity of the existential quantification is $2^{2|P|}$. The worse one of the two is $16|P|2^{4|P|}$. Therefore, the time spent on this step in the worst case is determined by $16|P|2^{4|P|}$.

Line 9. The time spent on this step is determined by the time spent on checking if PTY_1 is a subset of `states_seen`. Checking if PTY_1 is a subset of `states_seen` can be done by uniting PTY_1 and `states_seen` together and then seeing if the resultant set is equal to `states_seen`. Hence the time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 10. The time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 16. The time spent on this step in the worst case is determined by $16|P|2^{4|P|}$.

Line 17. The time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 18. The time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 26. This step repeats at most $2^{4|P|}$ times.

Line 27. This step repeats $|P|$ times.

Line 28. This step repeats $|A|$ times.

Line 29. The time spent on this step in the worst case is determined by $16|P| \times 2^{4|P|+1} + 2^{8|P|}$. Counting the larger one of the two, the time is determined by $2^{8|P|}$.

Line 30. The time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 31. The time spent on this step in the worst case is determined by $2^{8|P|}$.

Adding the time spent on each step together, we get $2^{4|P|} \times (|P| \times |A| \times (16|P|2^{4|P|} + 2^{8|P|} + 2^{8|P|} + 16|P|2^{4|P|} + 2^{8|P|} + 2^{8|P|}) + 2^{4|P|} \times |P| \times |A| \times (2^{8|P|} + 2^{8|P|} + 2^{8|P|}))$. Therefore, the complexity of the algorithm in the worst case is asymptotically bounded above by $3 \times 2^{16|P|}$. However, as the experimental data in Table 3 show, the situation in practice is far better than this.

6. Implementation

6.1. General description

AcPeg (Access Control Policy Evaluator and Generator) is implemented in Java. It integrates both the functions of the RW model-checking and the translation from RW to XACML (which is to be discussed in Section 7). A three-level abstraction mechanism is built into the tool to work with the RW model-checking. CEGAR (CounterExample-Guided Abstraction Refinement [11]) is also implemented in the tool, which can be used with the abstraction.

The translation works in Windows, Unix and Linux. The RW model-checking works only in Linux, because the BuDDy library used by AcPeg is for Linux only. However, one can run AcPeg as the RW model-checker in Windows provided one downloads the JavaBDD package for Windows from [31]. The BuDDy library for Windows can be found in that package. An explanation of the command line parameters for running AcPeg is in Appendix C.

6.2. Computational round

A computational round is a particular running of the algorithm when each quantified variable defined in the query is instantiated by an element in the class on which the variable is defined. In the query defined in Query 4.2, there are three quantified variables: a, c (disjointed) are defined on the set `Agent`, and p is defined on the set `Paper`. The set `Agent` is populated to four elements – $\{a_1, a_2, a_3, a_4\}$ – and the set `Paper` to three elements – $\{p_1, p_2, p_3\}$ – by the execution of the run-statement. Therefore the possible values that a can have is four, the possible values for c is four, and the possible values for p is three. The total number of combinations of the values of a, c and p is forty-eight. Each of the combinations, if run by the algorithm, becomes a round.

However, to obtain the overall result for checking a query, not every round is executed by the tool. The use of the keyword `disj` excludes those rounds where different quantified variables defined on the same class play the same element. Moreover, for an existential (universal) variable defined on a class, a round in which the checking result returned is true (false) excludes the necessity of running those rounds where only this variable is instantiated differently.

The above optimisations are built into the tool. However, the computation can be further simplified. In the cases that all quantified variables are made distinct using the keyword `disj`, every round returns the same result, and therefore the result returned by any round is the same as the overall result. This is so because in the cases that all quantified variables play different elements, the model built by the tool is symmetric. When running the tool, we leave the decision of running which round to the user (see Appendix C for the using of the option ‘r’). In the following discussions, all the experimental results on computational time and memory usage are the results obtained from running just one round.

6.3. Implementing strategies and guessing strategies

The RW program for Example 4.1 is shown in Fig. 4.

Now we write a query to ask the question that whether there is a strategy or guessing strategy for a to set z's value to false. This query is expressed by Query 6.1.

QUERY 6.1.

```
run for 1 P, 1 Agent
check{E p: P, a: Agent || {a}:{~z(p)}}
```

If we use option 'a' (see Appendix C) to tell AcPeg to search for strategies for Query 6.1, AcPeg finds none. Because to set z to false, a needs to be able to make either x's value true or y's value true. To have confidence to write the value of either x or y, a needs to find out u's value. However, u's value is unreadable by a. Therefore, there isn't a strategy which always guarantees that a can set z to false. But if we use option 'i' to run AcPeg to search for guessing strategies, the tool finds one, which assumes a can read every information she needs. The guessing strategy is shown in Fig. 5, where [p = 1 a = 1] stands for the fact that p is instantiated as the first element in the set `Paper` and a is instantiated as the first element in the set `Agent`. Strategies and guessing strategies are output to a file named 'strategy.acc' which is in the same directory with the checker.

```
AccessControlSystem exampleIntheSlide
  Class P;
  Predicate u(p: P), x(p: P), y(p: P), z(p: P);
  x(p){
      read: true;
      write: ~u(p);
  }
  y(p){
      read: true;
      write: u(p);
  }
  z(p){
      read: true;
      write: (x(p) or y(p));
  }
End
```

Fig. 4. The RW script for the policy defined in Example 4.1.

```
[p=1 a=1]
Acting agents: [1]
Guessing strategy: 1
if (u(1) is true) by 1 {
   set y(1) to true by 1;
   set z(1) to false by 1;
   skip;
}else {
   set x(1) to true by 1;
   set z(1) to false by 1;
   skip;
}
The number of guessing strategies found is: 1
```

Fig. 5. The guessing strategy found for Example 4.1.

6.4. *Abstraction*

To enable AcPeg to handle large cases, we added abstraction mechanisms to the tool. During the course of checking, the most time-consuming computations involve the BDDs which represent the conditions such as $V' = V \cup \{p_i\}$ in computing the pre-sets. Such conditions represent the fact that when an action, either sampling or overwriting, is performed on p_i, it only changes A's knowledge on p_i – it does not change A's knowledge on other variables in the set P. Using boolean expressions to represent such conditions is expensive. For example, $V' = V \cup \{p_i\}$ is represented by $(\bigwedge_{j=1,j\neq i}^{|P|}((v'_{pj} \wedge v_{p_j}) \vee (\neg v'_{pj} \wedge \neg v_{p_j}))) \wedge v'_{p_i}$. If $|P|$ is large, the length of the expression can be huge. For reasons of efficiency, we can choose to make this expression simpler by not maintaining A's knowledge of all the variables in P, when an action is performed only on p_i. For instance, if we only track A's knowledge of p_i when an action is performed on p_i, the above expression representing $V' = V \cup \{p_i\}$ becomes v'_{p_i}.

According to how many variables are tracked when modelling the change of A's knowledge when an action is performed on only one variable, we have introduced three abstraction levels in the tool for users to specify. The minimum level, level 0, is the level that no abstraction is used, that is, the tool maintains A's knowledge on all variables when actions are performed on any p_i. It is the most precise level and the default level, if no abstraction level is specified in the command line. The maximum level, level 2, is the level when an action is performed on p_i, the tool only maintains A's knowledge on p_i, and also on all the other variables that occur in ψ. In the middle, level 1 is built on the basis of level 2. In this level, the tool not only maintains A's knowledge on p_i and all the variables in ψ, as level 2 does, but also maintains A's knowledge on any other variables in P specified by the user in a configuration file named 'abstraction.config'. The more abstraction we use, from level 0 to level 2, the more precision we lose. If in level 1 or 2, the checking result is $\perp$, then it means there is no strategy for A to reach its goal. But if it is $\top$, it does not guarantee there is a strategy. In fact, the answer is uncertain. Because by not maintaining A's knowledge on all variables, some transitions which actually cannot happen may be included in the course of computing the pre-sets.

Working in abstraction, when a seemingly false strategy or guessing strategy is found, it is useful to know which variables in P, being not tracked, have caused the strategy or guessing strategy to be found. Once the variables are found, they can be put into the file 'abstraction.config' for AcPeg to track, so that, by running the tool once again, we can see whether the strategy or guessing strategy is found again. The repeating use of this procedure gradually rules out false strategies or guessing strategies found because of abstraction. This methodology is referred to as counterexample-guided abstraction refinement (CEGAR) in [11].

AcPeg provides a built-in function to do CEGAR. When option 'g' is specified, together with either abstraction level 1 or 2, AcPeg outputs a suggestion for each strategy or guessing strategy found. The suggestion is a list of variables in P suspected by AcPeg of being responsible for the strategy.

Consider the query defined in Query 4.2. We know that, if we use AcPeg to resolve the query without using abstraction, it finds no strategy. However, if we use AcPeg to resolve the query in abstraction level 2 together with CEGAR (running in Algo-0), we will see that the output contains a false strategy and a suggestion made by AcPeg, shown in Fig. 6.

The checking is performed in the round that a plays the first element of the set Agent, c plays the second and p is instantiated as the first element of the set Paper. The finding of this strategy is, as AcPeg suggests, because the variable author$(1, 1)$ is not tracked during the course of the checking. Following the suggestion provided by AcPeg, we put the variable author$(1, 1)$ into the file 'abstraction.config' by adding the line 'Predicate = author, Parameters = 1 1'. Then we run AcPeg again in abstraction level 1. This time no strategy is found.

```
[a=1 c=2 p=1]
Acting agents: [2]
Strategy: 1
set pcmember(1) to true by 2;
set reviewer(1,1) to true by 2;
skip;

Please track the following variable(s):
   - this may not be precise
author(1,1)

The number of strategies found is: 1
```

Fig. 6. A false strategy and a suggestion made by AcPeg when checking Query 4.2 in abstraction level 2 and CEGAR.

6.5. Strategies for nested queries

The strategy described by Strategy 1.1 is found by AcPeg when Query 4.3 is checked. The strategy is shown in Fig. 7.

The strategy described by Strategy 1.2 can be found if the following query is checked by AcPeg. (Query 6.2 differs from Query 4.3 only in that the condition reviewer(p, a)! is positive, meaning that initially a knows he is a reviewer of p.)

QUERY 6.2.

```
run for 1 Paper, 3 Agent
check {E disj a,b,c: Agent, p: Paper || chair(c)*! and
 ~author(p,a)*! and submittedreview(p,b)*! and
 ~submittedreview(p,a)! and pcmember(a)*! and
 reviewer(p,a)! and ~subreviewer(p,b,a)*! and
 ~subreviewer(p,c,a)*! and
 ~subreviewer(p,a,a)*! -> {a}:([review(p,b)] AND
 {a,c}:({submittedreview(p,a)}))}
```

However, as well as the strategy described by Strategy 1.2, AcPeg also finds a more straightforward one, where a first submits his review for p and then reads b's review for p. This is so because, along the way of achieving the first subgoal '[review(p, b)]', the second subgoal '{submittedreview(p, a)}' may also be achieved by a. The two strategies are shown in Fig. 8.

The goal in using AcPeg is to find undesirable strategies. If a query yields strategies which are not perceived as problematic, as in this case, the user should try to refine the query. To avoid the unwanted strategy in this case, one can refine the query by insisting that a achieves review(p, b) first, and then a, c together achieve submittedreview(p, a).

```
[a=1 b=2 c=3 p=1]
Strategy: 4.1
Coalition: [1]
if (review(1,2) is true) by 1 {
   skip;
}else {
   skip;
}

Coalition: [1, 3]
set reviewer(1,1) to true by 3;
set submittedreview(1,1) to true by 1;
skip;
```

Fig. 7. The strategy found for Query 4.3.

```
[a=1 b=2 c=3 p=1]
Strategy: 1.1
Coalition: [1]
set submittedreview(1,1) to true by 1;
if (review(1,2) is true) by 1 {
   skip;
}else {
   skip;
}

Coalition: [1, 3]
skip;

Strategy: 4.1
Coalition: [1]
set reviewer(1,1) to false by 1;
if (review(1,2) is true) by 1 {
   skip;
}else {
   skip;
}

Coalition: [1, 3]
set reviewer(1,1) to true by 3;
set submittedreview(1,1) to true by 1;
skip;
```

Fig. 8. Two strategies found for Query 6.2.

Query 6.3.

```
run for 1 Paper, 3 Agent
check {E disj a,b,c: Agent, p: Paper ||
        chair(c)*! and ~author(p,a)*! and
        submittedreview(p,b)*! and
        ~submittedreview(p,a)! and
        pcmember(a)*! and reviewer(p,a)! and
        ~subreviewer(p,b,a)*! and
        ~subreviewer(p,c,a)*! and ~subreviewer(p,a,a)*!
    -> {a}:(([review(p,b)] and {~submittedreview(p,a)})
           AND {a,c}:({submittedreview(p,a)}))}
```

This query does not yield the unwanted strategy.

The strategy found by AcPeg when Query 4.4 is checked, is shown in Fig. 9.

```
[a=1 c=2]
Strategy: 2.1
Coalition: [2]
set pcmember(1) to true by 2;
skip;

Coalition: [1]
set pcmember(1) to false by 1;
skip;

Coalition: [2]
set pcmember(1) to true by 2;
skip;

Coalition: [1]
set pcmember(1) to false by 1;
skip;

Coalition: [2]
set pcmember(1) to true by 2;
skip;
```

Fig. 9. The strategy found for Query 4.4.

6.6. *Case study – an employee information system*

EXAMPLE 6.1. An *Employee Information System* (EIS) is used to enforce authorisation rules on bonus allocation among the employees of a department. A bonus package with a fixed number of options, such as a free day off work, is available for all employees. The director of the department chooses options from the package to give to all employees. She can also read the information about the distribution of options. The director can promote an employee to be a manager. Managers can read and set ordinary employees' bonuses, but not bonuses of other managers or the director. An employee can appoint another employee to be his advocate, and have read access to his bonus information – for example, this might be useful if he needs help from a trade union.

To put this example in the RW formalism, let Bonus be the set of bonus options, Σ be the set of employees and thus P includes the following propositional variables, for all $b \in \texttt{Bonus}, a, a_1, a_2 \in \Sigma$:

bonus(a, b)	bonus option b is owned by a,
manager(a)	a is a manager in the department,
director(a)	a is the director of the department,
advocate(a_1, a_2)	a_2 is a_1's advocate.

The permission mappings $\mathtt{r}$ and $\mathtt{w}$ can be defined as follows:

$$\begin{array}{ll}
\mathtt{r}(\mathtt{bonus}(a,b),x) & \\
\quad\rightleftharpoons \begin{pmatrix} (x = a \vee \mathtt{director}(x)) \\ \vee\, (\mathtt{manager}(x) \wedge \neg\mathtt{manager}(a) \wedge \neg\mathtt{director}(a)) \\ \vee\, \mathtt{advocate}(a,x) \end{pmatrix} & \text{rule 1} \\
\mathtt{w}(\mathtt{bonus}(a,b),x) & \\
\quad\rightleftharpoons \begin{pmatrix} (\mathtt{manager}(x) \wedge \neg\mathtt{manager}(a) \wedge \neg\mathtt{director}(a)) \\ \vee\, \mathtt{director}(x) \end{pmatrix} & \text{rule 2} \\
\mathtt{r}(\mathtt{manager}(a),x) & \\
\quad\rightleftharpoons \mathtt{true} & \text{rule 3}
\end{array}$$

$$\begin{array}{ll}
\mathtt{w}(\mathtt{manager}(a),x) & \\
\quad\rightleftharpoons \big(\mathtt{director}(x) \vee (x = a \wedge \mathtt{manager}(a) \wedge \neg\mathtt{director}(a))\big) & \text{rule 4} \\
\mathtt{r}(\mathtt{director}(a),x) & \\
\quad\rightleftharpoons \mathtt{true} & \text{rule 5} \\
\mathtt{r}(\mathtt{advocate}(a_1,a_2),x) & \\
\quad\rightleftharpoons \mathtt{true} & \text{rule 6} \\
\mathtt{w}(\mathtt{advocate}(a_1,a_2),x) & \\
\quad\rightleftharpoons \big(x = a_1 \vee (\mathtt{advocate}(a_1,a_2) \wedge x = a_2)\big) & \text{rule 7}
\end{array}$$

The RW script for this example is shown in Fig. 10.

Rule 2 explicitly specifies that no manager can set another manager's bonus; otherwise two managers may co-operate to set each other's bonuses. However, the intention of rule 2 can be breached when several agents work together. Suppose $a_1, a_2 \in \Sigma$ are different agents; a_1 and a_2 are managers but not directors. Consider the query:

QUERY 6.4.

```
run for 4 Bonus, 8 Agent
check{E disj a1,a2: Agent, b: Bonus || ~director(a1)*!
    and ~director(a2)*!
    and manager(a1)! and manager(a2)! and ~bonus(a1,b) ->
 {a1,a2}:({bonus(a1,b)})}
```

For this query, AcPeg finds a strategy in which a_1 resigns his position as manager and then a_2 sets his bonus. Suppose we prevent this strategy by insisting that a_1 should be a manager at the end:

QUERY 6.5.

```
run for 4 Bonus, 8 Agent
check{E disj a1,a2: Agent, b: Bonus || ~director(a1)*!
```

```
AccessControlSystem EmployeeInformationSystem
 Class Bonus;
 Predicate bonus(employee: Agent, bonus: Bonus),
           manager(employee: Agent),
           director(employee: Agent),
           advocate(appointer: Agent,
                    appointee: Agent);
 bonus(a,b){
   read: (user=a or director(user))
          or (manager(user) and ~manager(a)
          and ~director(a)) or (advocate(a,user));
   write: (manager(user) and ~manager(a)
           and ~director(a)) or director(user);
 }
 manager(a){
   read: true;
   write: (director(user) or (user=a and manager(a)
           and ~director(a)));
 }
 director(a){
   read: true;
 }
 advocate(a1,a2){
   read: true;
   write: user=a1 or (user=a2 and advocate(a1,a2));
 }
End
```

Fig. 10. The RW script for the access control system in Example 6.1.

```
         and ~director(a2)*!
         and manager(a1)! and manager(a2)! and ~bonus(a1,b)
   -> {a1,a2}:({bonus(a1,b)} and {manager(a1)})}
```

Then AcPeg doesn’t find a strategy. Suppose now we introduce another agent a_3 which is a director, and ask the query

QUERY 6.6.

```
run for 4 Bonus, 8 Agent
check{E disj a1,a2,a3: Agent, b: Bonus || ~director(a1)*!
 and ~director(a2)*!
 and manager(a1)! and manager(a2)! and director(a3)*!
-> {a1,a2,a3}:({bonus(a1,b)} and {manager(a1)})}
```

For this query, we get the strategy in which a_3 directly sets the bonus of A_1, but we also get the strategy in which a_1 resigns her position as manager; then a_2 set a_1's bonus; and after that a_3 promotes a_1 to be manager again.

Indeed, if we suspect strategies like that one and want to check their possibility using AcPeg, we can do so. The following query directly finds the same strategy:

QUERY 6.7.

```
run for 4 Bonus, 8 Agent
check{E disj a1,a2,a3: Agent, b: Bonus || ~director(a1)*!
 and ~director(a2)*! and manager(a1)! and manager(a2)!
 and director(a3)*! -> {a1}:({~manager(a1)}
 AND {a2}:({bonus(a1,b)} AND {a3}:({manager(a1)}))))}
```

6.7. *Case study – a student information system*

EXAMPLE 6.2. A *Student Information System* (SIS) is a system which enforces authorisation rules for accessing students' marks of a particular module. The following rules apply:

1. Whether an agent is a student is readable by all the agents.
2. Whether an agent is the lecturer of the module is readable by all the agents. There is only one lecturer.
3. Whether a student's year is higher than another student's is readable by all the agents.
4. Whether a student is a demonstrator of another student is readable by all the agents.
5. The lecturer can appoint a student in a higher year to be a demonstrator of a student in a lower year.
6. Whether a student can write another student's mark is readable by the former.
7. The lecturer can give writing permissions to a demonstrator.

To model this example in the RW formalism, let Σ be the set of agents and thus P includes the following propositional variables, for all $a, a_1, a_2 \in \Sigma$:

student(a)	a is a student,
lecturer(a)	a is the lecturer,
higher(a_1, a_2)	student a_1 is in a year higher than that of student a_2,
demonstrator_of(a_1, a_2)	student a_1 is a demonstrator of student a_2,
mark(a)	student a's mark. (Here, in a highly abstract way, we think of a student's mark as either *pass* or *fail*.)

```
AccessControlSystem StudentInformationSystem
 Predicate lecturer(agent: Agent)!,
           student(agent: Agent),
           demonstrator_of(demonstrator: Agent,
                           student: Agent),
           higher(senior: Agent, junior: Agent),
           mark(student: Agent);
 lecturer(l){ read: true;
 }
 student(s){ read: true;
 }
 higher(s,j){ read: true;
 }
 demonstrator_of(d,s){
   read: true;
   write: (lecturer(user) & higher(d,s)) |
          (demonstrator_of(d,s) & user=d);
 }
 mark(a){
   read: user=a;
   write: lecturer(user) | demonstrator_of(user,a);
 }
End
```

Fig. 11. The RW script for the access control system in Example 6.2.

We assume that the relation `higher` is antisymmetric and transitive. Note, however, that currently there is no way in the RW language to specify such constraints. (See 8.2.2 for the discussion on the limitation of the RW language on specifying integrity constraints.) This means that AcPeg could find attack strategies based on impossible interpretation of the `higher` relation. The user needs to be aware of this.

The RW script for the model is in Fig. 11.

In such a system, it is important to ensure that no two students can write each other's marks. Otherwise, they may conspire to increase their marks. To verify this property we can check Query 6.8, which asks that 'can the lecturer appoint two students to be demonstrators of each other, and as a result, they can write each other's marks.'

QUERY 6.8.

```
check{E disj l,a1,a2: Agent || lecturer(l)*!
 and student(a1)*! and student(a2)*!
 and higher(a1,a2)*! -> {l}:{demonstrator_of(a1,a2)
 and demonstrator_of(a2,a1)}}
```

The query was checked by both Algo-0 and Algo-1. Both found no strategy.

6.8. *Case study – a patient record system*

The policy defined in this example is based on the case study of the *Multidomain Healthcare System* in [5].

EXAMPLE 6.3. A *Patient Record System* (PRS) is a system which enforces authorisation rules for accessing patients' electronic health records (EHRs). A patient may explicitly exclude certain individuals from accessing her EHR. Doctor d of a hospital may access the record of patient p if d is a treating doctor of p and d is not in p's exclusion list. Doctor d becomes a treating doctor of patient p if a nurse-on-duty assigns p to d while d is on duty. As soon as d ceases to be a treating doctor of p, he can no longer access p's record.

To model this example in the RW formalism, let Σ be the set of agents and thus P includes the following propositional variables, for all $a, a_1, a_2 \in \Sigma$:

patient(a)	a is a patient,
doctor_on_duty(a)	a is a doctor-on-duty,
nurse_on_duty(a)	a is a nurse-on-duty,
excluded(a_1, a_2)	a_2 is on the exclusion list of patient a_1,
record(a)	patient a's record (Here, in a highly abstract way, we think of a patient's record only containing one bit of information.),
treating_doctor(a_1, a_2)	a_1 is a treating doctor of patient a_2.

The RW script for the model is in Fig. 12.

In this example, we check that whether a treating doctor having ceased to be a treating doctor can re-gain the privilege of overwriting a patient's record without the help of other agents. The property is expressed by Query 6.9.

QUERY 6.9.

```
run for 6 Agent
check{E disj p,d: Agent || treating_doctor(d,p)!
 and patient(p)*!
 -> {d}:({~treating_doctor(d,p)}
 AND {d}:({record(p)})))}
```

Query 6.9 was checked by both Algo-0 and Algo-1 and no strategy was found.

```
AccessControlSystem PatientRecordSystem
 Predicate patient(agent: Agent), doctor_on_duty(agent: Agent),
           nurse_on_duty(agent: Agent),
           excluded(patient: Agent, agent: Agent),
           record(patient: Agent),
           treating_doctor(doctor: Agent, patient: Agent);
 patient(a) { read: true;
 }
 doctor_on_duty(a) { read: true;
 }
 nurse_on_duty(a) { read: true;
 }
 excluded(p,a) { read: user=p; write: user=p;
 }
 record(p) {
   read: user=p | (~excluded(p,user) & treating_doctor(user,p));
   write: treating_doctor(user,p) & ~excluded(p,user);
 }
treating_doctor(d,p) {
  read: true;
  write: (nurse_on_duty(user) & doctor_on_duty(d) & patient(p)
           & ~treating_doctor(d,p))
         | treating_doctor(d,p);
 }
End
```

Fig. 12. The RW script for the access control system in Example 6.3.

6.9. *Performance*

We discuss the performance of the tool in terms of the memory it uses and the time it spends on a checking. Two main factors influence its performance: the nature of a query and the number of variables in P (see the discussion in Section 5.9).

In Table 2, we summarise the data on the memory usage and the time spent on all the testing cases discussed in the previous subsections of this chapter. All the checks are performed without using abstraction. The computer used is a laptop running Linux (kernel 2.6.10) on a Pentium M 1.6GHz and 512MB RAM. The table includes results from both Algo-0 and Algo-1, so that the performance of the two variants of the algorithm can be compared. The general performance of the tool is demonstrated by the experimental results. The unit for memory usage is MB (mega byte) and the unit for time spent is ms (millisecond).

A general observation is that the memory usage does not increase as dramatically as the time spent when the size of the model increases. In terms of memory usage, there is no observable difference between Algo-0 and Algo-1. However, as for the computational time, we may conclude that the performance of Algo-1 is slightly better than that of Algo-0 (see also Table 3).

Now we shall rule out the influences of the queries and observe how the increasing of the size of a model solely affects the tool's performance. To do this test, we use

Table 2

Experimental results obtained from running the test cases in Section 6

Query	Assignment	$\|P\|$	Memory usage		Time spent	
			Algo-0	Algo-1	Algo-0	Algo-1
Query 4.2	Paper = 3 Agent = 4	104	156	156	3600	3600
Query 4.3	Paper = 1 Agent = 3	27	155	155	250	250
Query 6.2	Paper = 1 Agent = 3	27	155	155	1000	700
Query 4.4	Paper = 1 Agent = 3	27	155	155	250	252
Query 6.4	Bonus = 4 Agent = 8	112	162	162	30000	22500
Query 6.8	Agent = 10	230	156	156	33807	34181
Query 6.9	Agent = 8	160	156	156	41979	41284

Table 3

Experimental results obtained from checking Query 6.4 with different sizes of the model

Assignment	$\|P\|$	Memory usage		Time spent	
		Algo-0	Algo-1	Algo-0	Algo-1
Bonus = 3 Agent = 3	24	155	155	324	288
Bonus = 3 Agent = 5	50	156	156	1410	1024
Bonus = 4 Agent = 6	72	159	159	3326	2595
Bonus = 4 Agent = 8	112	162	162	30000	22500
Bonus = 5 Agent = 10	170	169	169	81000	58660
Bonus = 6 Agent = 12	240	195	195	226764	177357

Query 6.4 and assign different elements to the set Bonus and the set Agent. The experimental results are summarised in Table 3.

From the data in Table 3 we can see that, in practice, the computing time increases far slower than the estimation under the worst case discussed in Section 5.9. Two more reasons make the prospect of using AcPeg even more optimistic. As Daniel Jackson has argued in the case of Alloy, small scope checks are extremely valuable for finding errors [23]. Most errors can be found by checking models of small sizes. Moreover, the tool outputs a strategy whenever it finds one. Therefore one does not need to wait to the end of a checking. As soon as a strategy is output one can abort the program immediately.

7. RW to XACML

The *eXtensible Access Control Markup Language* (XACML) is an access control policy specification language created by the OASIS committee [15]. It is intended to be used as a standard language in the field of e-business. Policies written in XACML can be bolted onto existing access control systems. Or, new access control systems can be implemented based on existing XACML policies. However, policies written in XACML are hard to read and analyse directly. The syntax of XACML is neither compact nor readable to someone who is not familiar with it. Writing policies in XACML directly by hand is not only difficult but also error-prone. However, AcPeg makes this task easier by translating policies written in the RW language, which offers a compact, readable syntax, into XACML. This is possible mainly because, like authorisation conditions in RW, XACML represents authorisation conditions using the form of boolean expressions. The benefit of the translation is twofold: First, it allows the relatively concise descriptions of access control policies in RW to be automatically translated into the machine-oriented XACML. Secondly, since the properties of systems described in RW can be verified algorithmically, the translation guarantees to produce policies which preserve the required properties.

The XACML we generate from an RW script also includes some SQL. This is used to access a database during run-time, in order to allow state information to be used in taking access control decisions.

Sun Microsystems has implemented a set of Java classes [30] for XACML. It includes a program which simulates the functions of a policy decision point, that is, it reads XACML policies; searches for values for necessary attributes and makes decisions. In this way, one could "bolt on" access control classes to an existing Java program, by writing the policy in RW, then using our translator to XACML, and then generating Java classes using Sun's compiler.

We discuss the translation by first giving a little background knowledge about XACML.

7.1. Background on XACML

7.1.1. Data-flow model of XACML

An access control system using XACML as its policy specification language is meant to be used on the Internet, where different components of the system locates throughout the network. The data-flow model which describes how information is exchanged between the components is shown in Fig. 13.

Access control policies written in XACML are stored in the policy administration point (PAP). This PAP is known to the policy decision point (PDP), which is the entity that makes access decisions. The policy enforcement point (PEP) is the entity which implements and enforces mechanisms of access control. When it receives a request, it passes the request to the context handler. The context handler then assembles the request into a format specified by XACML and passes it to the PDP. On receiving the request, the PDP searches through the policies provided by the PAP and picks up an applicable policy, if there is one, and makes a decision based on the policy and the content of the request. To make the decision, the PDP may need to consult the context handler to find out values of certain attributes which are necessary to make the decision. The context handler will gather all that information from different sources,

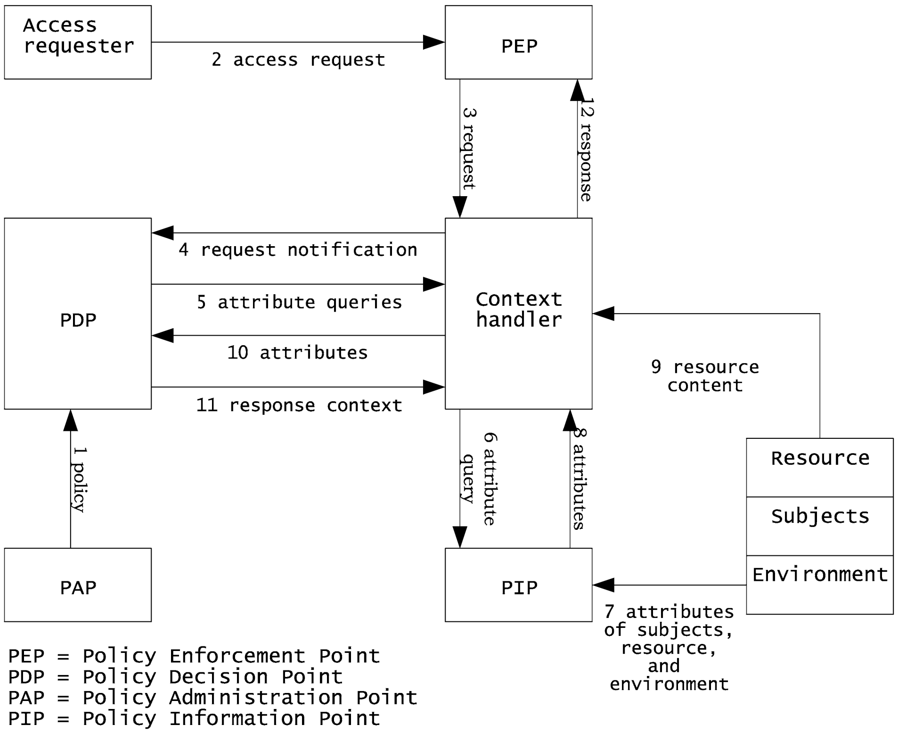


Fig. 13. The data-flow model of XACML.

such as from the policy information point (PIP), from the environment, from the subjects and resource. Once a decision is made, the PDP will send it back to the context handler, who will transform the response into a format understandable to the PEP and forward it to the PEP.

7.1.2. Rule, policy and policy-set

The most basic functional unit in XACML is a *rule*. A number of rules form a *policy*. A number of policies form a *policy set*. A complete rule consists of a *head*, a *description*, a *target* and a *condition*. The head contains a XML name space declaration, a name for the rule, and the effect of the rule, either *Deny* or *Permit*. The description describes the rule in human languages, and thus makes the rule more understandable. The target defines applicable situations for the rule. If the target is evaluated to false, the rule will be simply rendered as *not applicable* and the condition will not be considered. The condition represents a boolean expression, just as the target, which refines the applicability of the rule. Only if the target and the condition are both evaluated to true, is the effect of the rule returned. Otherwise this rule is reckoned as *not applicable*.

The structure of a policy is very much like that of a rule. It contains a *head*, a *description* about the policy, a *target* defining the applicability of the policy, and a number of rules. However, in the policy, a *rule-combining algorithm* must be specified to resolve conflicting results returned by different applicable rules. For example, if the *deny-overrides* algorithm is used, the effect is that if any rule is evaluated to *Deny*, the policy must return *Deny*. The *rule-combining algorithm* is specified in the head of the policy.

Likewise, the structure of a policy set is like that of a policy, except that a policy set uses a *policy-combining algorithm*.

7.2. Compiling RW to XACML and SQL

7.2.1. Structure of the converted XACML file

The compiler reads a RW file and outputs the corresponding XACML file. The XACML file is a single policy unit which contains a number of rules. Each conditional formula in the RW file generates a rule in XACML plus a default rule which denies everything. The structure of the output XACML file is shown in Fig. 14.

The title of the policy is composed of the content specified in the specification of XACML 1.1, but one can make one's own modifications to suit one's own need. We chose the *permit-overrides algorithm*. This is one of the algorithms that are used in XACML to reconcile decisions from multiple applicable rules. The algorithm approves *Permit*, provided that at least one of the applicable rules does so. If some rules produce *Deny* and all other rules produce *NotApplicable*, the algorithm approves *Deny*. In other words, *Permit* takes precedence.

The target of the policy is made to apply to every situation. No target applicability constraint is needed at the policy level, because each rule defines its applicability under its own *Target* tag. Rules are placed after the *Target* tag of the policy. The effect of all rules, except the last default one, is *Permit*. Together with the *permit-overrides* algorithm, the policy denies whatever is not explicitly permitted.

For the *conference paper review system* defined in Example 3.1, the generated XACML policy contains thirteen rules, including a default denying rule at the end. Each single rule is generated from a conditional formula in the RW script. An example of this correspondence in the case of the formula defining the reading privilege for the parameterised predicate `pcmember`(a) is shown in Fig. 15. The rule contains a *Target* tag, which evaluates applicable situations for this rule.

The *Subjects* tag of the *Target* section sets no restriction on the criterion of the requester, and this is the case with all rules generated by AcPeg.

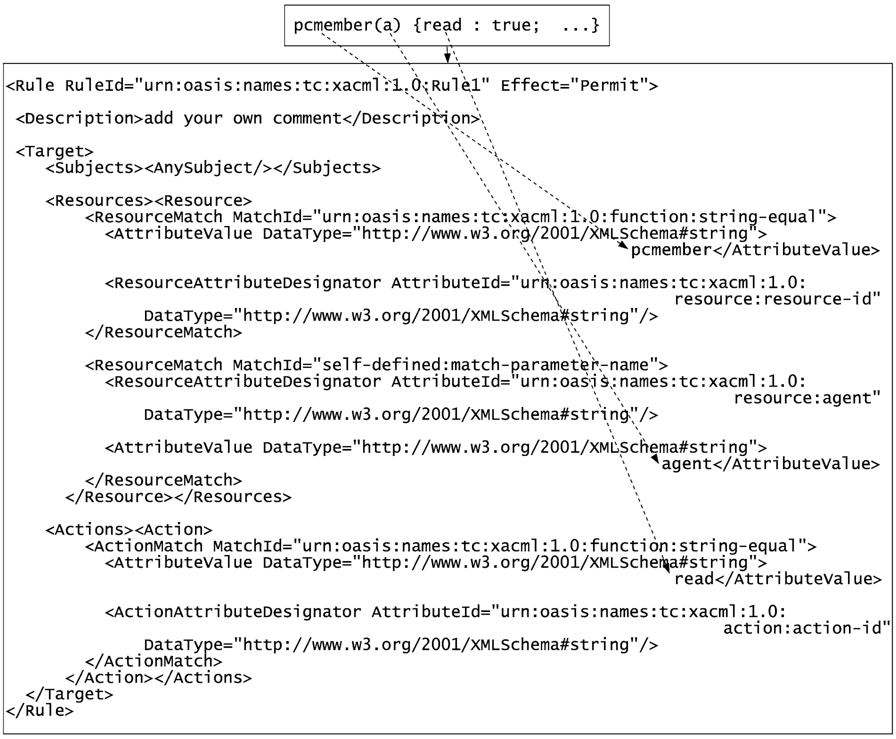
The *Resources* tag has two criteria to be evaluated (see Fig. 15). The first one is whether the name of the requesting resource is the same as the predicate name `pcmember`. To evaluate this criterion, a PDP program, following the instruction in the *ResourceAttributeDesignator*, will select the attribute value from the *resource-id* field in a request (send on link 4 in Fig. 13), shown in Fig. 16.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns= ...
PolicyId="conference"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
              rule-combining-algorithm:ordered-permit-overrides">

<Description>add your own comment</Description>

<Target>
  <Subjects><AnySubject/></Subjects>
  <Resources><AnyResource/></Resources>
  <Actions><AnyAction/></Actions>
</Target>
...
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule8" Effect="Permit">
  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources>...</Resources>
    <Actions>...</Actions>
  </Target>

  <Condition ...>
  ...
<\Condition>
<\Rule>
...
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule11" Effect="Deny">

  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions><AnyAction/></Actions>
  </Target>
</Rule>
</Policy>
```

Fig. 14. The structure of an output XACML policy file.

If the name of the requesting resource is not `pcmember`, this rule is simply evaluated as *not applicable*. The second criterion is whether the name of the parameter whose type is `Agent` is `agent`. This information is retrieved from the field *agent* in the request. The value retrieved, in this case, will be `agent` $=$ $\mathtt{a}_1$, which specifies the name of the parameter and (‘$=$’) its actual value. Here we use a dedicated external function (that is, a function written by us) which compares the string selected from the XACML policy, which, in this case, is `agent`, with the string on the left side of the equivalent formula, which is expected to be `agent` too. The two criteria are enclosed in one *Resource* tag, which means the conjunction of these criteria. For the evaluation to be successful, both of these criteria must be met.

The *Actions* tag is to evaluate whether the attribute value of the *action-id* field in the request matches the applicable action. Since the conditional formula applies to



```
pcmember(a) {read : true;  ...}
```

```
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule1" Effect="Permit">

 <Description>add your own comment</Description>

 <Target>
    <Subjects><AnySubject/></Subjects>

    <Resources><Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                                                        pcmember</AttributeValue>

          <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                             resource:resource-id"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </ResourceMatch>

        <ResourceMatch MatchId="self-defined:match-parameter-name">
          <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                                   resource:agent"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>

          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                                                            agent</AttributeValue>
        </ResourceMatch>
      </Resource></Resources>

    <Actions><Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                                                             read</AttributeValue>

          <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                                action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </ActionMatch>
      </Action></Actions>
  </Target>
</Rule>
```

Fig. 15. Correspondence between a privilege definition and its generated rule.

the privilege of reading in this example, the applicable action for this rule is `read`. Since the condition for reading in the RW file is `true`, the *Condition* tag is omitted in the XACML file. However, if a condition in the RW file is non-trivial, a *Condition* tag is added to the generated rule. We explain how such tags are generated in what follows.

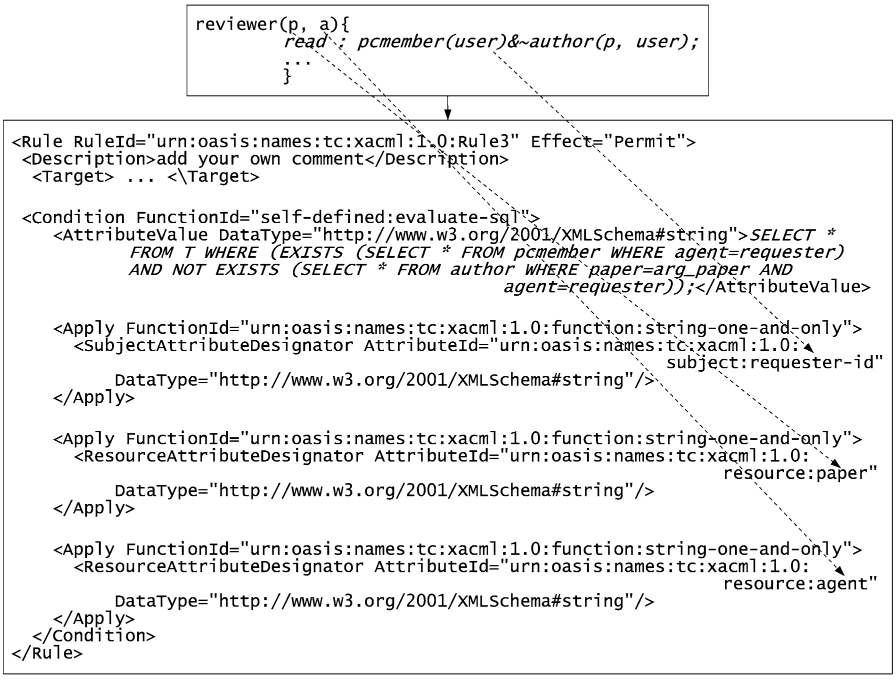
7.2.2. Generating Condition tag

Logical formulas defining reading and writing privileges in a RW script are converted into a SQL statement and put under *Condition* tags. The conditions under a *Condition* tag are called to be evaluated only if the target-evaluation is passed. A SQL statement is evaluated by calling a dedicated external function, which queries a database storing values of the attributes in the system. To understand the idea, consider the RW formula and its generated conditions shown in Fig. 17.

AcPeg produces SQL code that looks up the truth values of the variables in P in a database. For instance, given an $a \in$ `Agent`, the truth value of `pcmember`(a) can be determined by looking up a in a table which contains all the agents who possess the role PC member. By this way, any boolean combination of such variables can be evaluated. Thus, the truth value of a RW formula is reflected by the result of querying

```
<?xml version="1.0" encoding="UTF-8"?>
<Request
      xmlns="urn:oasis:names:tc:xacml:1.0:context"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:context
      cs-xacml-schema-context-01.xsd">

  <Subject> <AnySubject/></Subject>

  <Resource>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                  <AttributeValue>pcmember</AttributeValue>
          </Attribute>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:agent"
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                  <AttributeValue>agent=a1</AttributeValue>
          </Attribute>
  </Resource>

  <Action>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                  <AttributeValue>read</AttributeValue>
          </Attribute>
  </Action>
</Request>
```

Fig. 16. A request for the rule in Fig. 15.



```
reviewer(p, a){
        read : pcmember(user)&~author(p, user);
        ...
        }
```

```
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule3" Effect="Permit">
 <Description>add your own comment</Description>
  <Target> ... <\Target>

 <Condition FunctionId="self-defined:evaluate-sql">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">SELECT *
           FROM T WHERE (EXISTS (SELECT * FROM pcmember WHERE agent=requester)
           AND NOT EXISTS (SELECT * FROM author WHERE paper=arg_paper AND
                                                  agent=requester));</AttributeValue>

    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                                subject:requester-id"
          DataType="http://www.w3.org/2001/XMLSchema#string"/>
    </Apply>

    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                                  resource:paper"
          DataType="http://www.w3.org/2001/XMLSchema#string"/>
    </Apply>

    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                                  resource:agent"
          DataType="http://www.w3.org/2001/XMLSchema#string"/>
    </Apply>
  </Condition>
 </Rule>
```

Fig. 17. A formula in a RW script and its generated conditions.

its converted SQL statement. The external function *self-defined:evaluate-sql*, which appears as a conditional function in *Condition* tags, evaluates SQL statements.

We now define the mapping SQL from RW formulas to SQL statements, implemented by AcPeg. The translation is motivated by the correspondence between first-order logic and SQL (see [1]).

SQL(pcmember(a)) $\rightleftharpoons$ EXISTS (SELECT * FROM pcmember WHERE agent = arg_agent) This is an example for the simplest case, in which the condition contains just one predicate with one parameter. Then the condition is equivalent to the non-emptiness of the selection result. The column name *agent* is derived from the definition of the predicate pcmember(agent : Agent). Thus the table *pcmember* must contain a column named *agent*. The string *arg_agent* is the formal name of parameter a. The actual value of this parameter comes from the request. The external function replaces formal names by their actual values and produces an executable SQL statement.

SQL(R($a_1, \ldots, a_n$)) $\rightleftharpoons$ EXISTS (SELECT * FROM R WHERE $\mathrm{R}_{c1} = a'_1$ AND $\ldots$ AND $\mathrm{R}_{cn} = a'_n$) If the predicate has more than one parameter, the SQL selection condition is a conjunction. Again, the RW condition is equivalent to the non-emptiness of the selection result. Here $\mathrm{R}_{c1} \ldots \mathrm{R}_{cn}$ stand for the column names derived from the definition of predicate R. $a'_1 \ldots, a'_n$ are the formal names for $a_1 \ldots, a_n$.

SQL($a_1 = a_2$) $\rightleftharpoons$ ($a'_1 = a'_2$) The translation of equality formulas in RW is straightforward. a'_1 and a'_2 are the formal names for a_1 and a_2.

SQL($\neg f$) $\rightleftharpoons$ NOT SQL(f), SQL($f_1 \wedge f_2$) $\rightleftharpoons$ (SQL(f_1)) AND (SQL(f_2)) Logical operators are expressed by their counterparts in SQL. We only give the clauses for negation and conjunction. Other connectives can be defined using these.

SQL($\exists\, x \in \mathrm{D}\, f$) $\rightleftharpoons$ EXISTS (SELECT id FROM D AS D_i WHERE SQL(f)) Elements are selected from the table D, which is supposed to list all the elements from the RW class with the same name, and f is evaluated for each of them. The RW condition $\exists\, x \in \mathrm{D}\, f$ holds if and only if the resulting selection is non-empty. The string *id* is the default name for a column in tables describing defined classes. The alias D_i is given to D to avoid clashes between names of bound variables. Universal formulas are expressed using existential ones by means of negation.

To obtain a complete translation of a RW formula, the result of SQL is prefixed **SELECT * FROM T WHERE**. Here T is a purpose-set table which is just supposed to contain an appropriate string to be returned by the external function which evaluates SQL statements. Thus the final form of the converted SQL statement for a given f is **SELECT * FROM T WHERE (**SQL(f)**);**.

The above clauses allow any RW formula to be translated into an SQL statement. Fig. 17 shows an example. The string *requester* is the formal name for the access requester. It becomes replaced by its actual value by the external function.

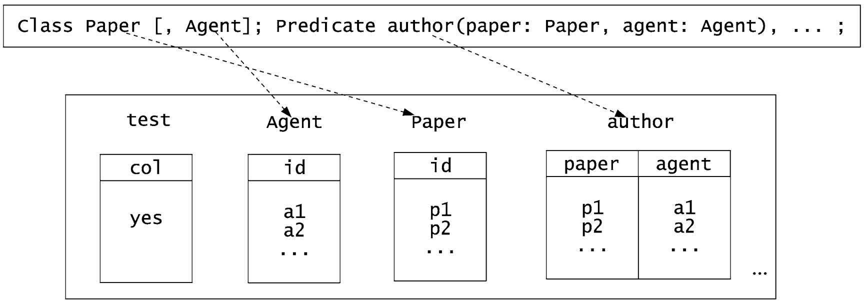
7.2.3. *The external function – self-defined:evaluate-sql*

The external function, *self-defined:evaluate-sql*, is called by a PDP program to read an SQL statement and other parameters; replace formal names in the SQL statement by their actual values; execute the query and return the result, either `true` or `false` to the PDP program. It takes at least two parameters, which are the SQL statement and the actual value for the requester selected from the request. AcPeg also puts all the formal names that need to be resolved after the SQL statement and the *SubjectAttributeDesignator* on the requester, as Fig. 17 shows. The *ResourceAttributeDesignator* on `paper` is to select the actual value for the first parameter of predicate `reviewer` – p, and the *ResourceAttributeDesignator* on `agent` is to select the actual value for the second parameter of predicate `reviewer` – a. These two values are passed to the external function. The external function uses the value selected by the *SubjectAttributeDesignator* for *requester-id* to replace every occurrence of the string *requester* in the SQL statement, the value selected by the *ResourceAttributeDesignator* on `paper` to replace every occurrence of the string *arg_paper* and the value selected by the *ResourceAttributeDesignator* on `agent` to replace every occurrence of the string *arg_agent*. Strings in the request are written *without* quotation marks. These are added by the external function, because quotation marks are required by SQL. The last formal name to be replaced is T. It becomes replaced by the actual name of the table, which is *test* in our example.

7.3. *Assumptions about the database*

The database must satisfy some conditions to make the translation work. Fig. 18 illustrates the idea of how we set up the database for the conference example. Table *test* corresponds to the virtual table T. Table *Agent* and *Paper* correspond to the defined class `Agent` and `Paper`. Table *author* is the table derived from the predicate `author(paper : Paper, agent : Agent)`. Tables derived from other predicates are not shown in the figure. The elements in the tables are only for the purpose of illustration.

The conditions are given as follows:



```
Class Paper [, Agent]; Predicate author(paper: Paper, agent: Agent), ... ;
```

test

col
yes

Agent

id
a1 a2 ...

Paper

id
p1 p2 ...

author

paper	agent
p1 p2 ...	a1 a2 ...

Fig. 18. Tables and the structure of the database set up for the conference example.

- The database must contain a table for each defined class and defined predicate, including a table for class `Agent` and a table for T, except for the defined predicates that do not appear in any of the logical formulas.
- If a table is for a defined class, it must have a column named *id* to store identifiers of the elements that can be used to uniquely identify each individual element in that class. The type of the column must be either *string* or *char*. The table must not contain duplicated records for each individual element in the class. One can store any other information about the elements in that table, however, no matter what they are, they will not be queried during an evaluation.
- If a table is for a defined predicate, it must contain a column corresponding to each of its parameters, bearing the same name of that parameter. One can store other information in the table, but it will not be queried.
- The table for T needs only one column and it does not matter what the type and the name of the column are. It needs one record which can have whatever value. However, if one is to give it a name differing from *test*, one should modify the source code of the external function that evaluates SQL statements. The modification is very simple. One only needs to change the definition of the *String* variable that stores the value for the name of the table. That variable is named *test*.
- In our example, the name for the database is *conference*. This has been hard-coded into the external function to make it work. One needs to modify the variable *url* in the source code of the function, which is type of *String*, if one's database has another name. One should also modify the location of the database stored in that variable. One also needs to change the driver for the database in the code, if one uses a database system other than *Postgresql*. The driver information is stored in the *String* variable *driver*.

8. Discussions and conclusions

8.1. Summary

In this paper we demonstrate the applicability of using the RW framework to uncover security weaknesses in access control policies and to generate verified policies. The particular security weaknesses that the RW framework intends to uncover are potential failures in policies caused by *interactions of rules*, *co-operation between agents* (*including changing of each other's privileges*) and *multi-step actions*. As we have mentioned, security weaknesses caused by these reasons are hard to be found by traditional approaches in the field of security policy analysis. The RW framework uses model-checking to address these problems. In the framework, the RW formalism is used to model access control policies; the RW language is used to express models in RW as well as properties to be verified; and AcPeg is used to verify whether a model satisfies a property as well as to perform the translation from RW to XACML.

We formally defined the RW formalism. Using an example, we showed how to model an access control policy in the RW formalism. We discussed issues on the syntax and the semantics of the RW language, especially on how to write queries. We presented the RW model-checking algorithm; proved its correctness; and discussed its computational complexity. The usability of the algorithm was demonstrated through a series of case studies in which queries are solved by AcPeg. Functions of AcPeg and abstraction mechanisms were explained. Finally, the translation from RW to XACML was discussed.

8.2. Practical applicability of the RW framework

8.2.1. The modelling power of the RW formalism

The practical applicability of the RW framework first depends on the modelling power of the RW formalism. Although it is hard to express law within logic [24], the RW formalism can be used to model policies used by a wide variety of access control systems. As the case studies in Section 3 and Section 6 have shown, the RW formalism is suitable to model systems adopting either role-based or individual-based policies. Given an access control system, what the RW formalism models are data of the system, relations between data and permissions (which are considered as data as well). All these are represented by propositional variables in the RW formalism. When modelling a system in the RW formalism, one should avoid defining too many predicates. A large number of predicates may cause an instancing model to have too many variables and thus a checking upon the model takes too long.

8.2.2. The expressive power of the RW language

The practical applicability secondly depends on the expressive power of the RW language. One major limitation of the RW language that we have identified through our experience with it is that, currently, there is no way to explicitly establish integrity constraints between the values of different variables.

For example:

- In the RW language, there is no way to express mutual exclusions between the values of different variables. For instance, in Example 6.2, we cannot express the rule ‘an agent cannot be a student and a lecturer at the same time’, that is, for an agent a, the value of $\texttt{student}(a)$ being true excludes the possibility of $\texttt{lecturer}(a)$ being true. Although in Example 6.2, to include this rule is not absolutely necessary, still, it is better if we could express it to make the model more precise.
- In the RW language, there is no way to express inheritance between roles. In the context of the RW formalism inheritance means that one variable’s value being true implies another’s being true. For example, in Example 3.1, for an agent a, the value of $\texttt{chair}(a)$ being true implies the value of $\texttt{pcmember}(a)$ being true. Inheritance reflects the hierarchical structures in the organisation of many entities in the real world. The ability to express inheritance therefore is useful to the RW language.

8.2.3. The checking ability of AcPeg

The practical applicability thirdly depends on the checking ability of AcPeg, which is subject to the threat of the state explosion problem. From experimental experiences we have found that the time spent on a checking not only depends on the size of the model but also on the content of the query. Generally speaking, the more variables are used in conditions to qualify the agents' knowledge states the faster the checking is. The reason is that starting from a knowledge state where the agents already have some knowledge about the system will take them less steps to reach the goal than starting from a knowledge state where they know nothing, and thus it will take AcPeg less time spent on figuring out the strategy. Moreover, AcPeg tends to spend longer time on goals that can be achieved in many ways than goals that can be achieved in only a few ways. In some circumstances where there are more than one acting agent, the sequence of the agents that appears in the coalition may also influence the length of the time spent on a checking. Finally, it matters which variant of the algorithm is used – Algo-0 or Algo-1. A general observation is that, for a given query, Algo-1 tends to be faster in checking than Algo-0 if no abstraction is used. Neither with Algo-0 nor with Algo-1 can we control which strategies are found and which are not.

8.3. Amending policies

The output strategies help to find out how a goal can be achieved. In the case that a goal should not be achieved, the output strategies give us ideas as to how the policy should be amended to make the goal unachievable.

The strategy in Fig. 7 reveals that, as a PC member, a could have already read b's review for p before c assigns p to a for reviewing. The second strategy in Fig. 8 shows that a could resign his reviewership for p; read b's review for p; and then be assigned p for reviewing by c again. The reason that the strategy in Fig. 7 exists lies in rule 11 defined in Example 3.1. It does not constrain the reading privilege on a review to reviewers, and thus provides opportunities for non-reviewers to read reviews. To amend the rule we could add a condition to the rule which states that to read a review one must be a reviewer. Having added the condition to the formula in the RW script, the amended rule is shown in Fig. 19, where the condition is added at the end.

The reason that the second strategy in Fig. 8 exists lies in the fact that the policy defined in Example 3.1 allows a PC member to be assigned a paper that was previously assigned to him. To prevent this from happening, an additional rule might be added to the policy, stating that a PC member cannot be assigned a paper that was once assigned to him, even if he later resigned being reviewer of the paper. To express this rule in the RW language, we need to define an additional predicate to denote the fact that a paper was once assigned to a PC member. For $a \in \texttt{Agent}$, $p \in \texttt{Paper}$, we have:

$\texttt{assigned}(p, a)$ p was once assigned to PC member a for reviewing

```
review(p, a){
   read : pcmember(user) & ~author(p, user)
            & submittedreview(p, a)
            & (((reviewer(p, user)
               -> submittedreview(p, user))
              and (E b: Agent [subreviewer(p, b, user)]
                   -> submittedreview(p, user)))
                | user=a) & (E p1: Paper [reviewer(p1,user)]);
   write : ...;
}
```

Fig. 19. The amended version of rule 11 in Example 3.1.

```
reviewer(p, a){
   read : ...;
   write : (chair(user) & pcmember(a) & ~author(p,a)
           & ~assigned(p,a)) or ...;
        }
```

Fig. 20. The amended version of rule 3 in Example 3.1.

```
assigned(p, a){
   read : true;
   write: chair(user) & ~assigned(p,a) & reviewer(p,a);
        }
```

Fig. 21. The accessing rules for the predicate assigned(paper : Paper, agent : Agent)!.

We then change rule 3 to 'The PC chair can assign a paper to a PC member for reviewing, provided the PC member is not the paper's author and the paper was not once assigned to the PC member'. The amended rule is shown in Fig. 20.

For access rules to the predicate assigned(paper : Paper, agent : Agent), see Fig. 21. The overwriting rule says that once a paper p is assigned to a PC member a, assigned(p, a) is marked as true and cannot be changed to false afterwards.

Based on all the improvements discussed in this section, a new model for the policy which applies to the conference paper review system in Example 3.1 is constructed. Now, checking Query 4.3 (the condition $\neg$assigned(p, a)! is added) on the new model, AcPeg finds no strategy (both in Algo-0 and Algo-1). Checking Query 6.2 on the new model, AcPeg finds one strategy (both in Algo-0 and Algo-1) which requires a to submit her review for p before reading b's review for p.

8.4. *On the translation to XACML*

It should be noted that our translation from RW formulas, which are essentially first order formulas interpreted on a finite model, to SQL statements, was chosen for its simplicity and is far from optimal. There is extensive literature on relational database query optimisation and, in particular, on the correspondence between relational and first order queries, see, e.g. [9].

The verification methods for RW apply to models of access control systems with fixed sets of agents and resources only. However, this limitation does not apply to the translation to XACML of such access control systems as described in this paper. RW permission conditions written using quantifiers can meaningfully apply to systems with varying sets of resources and agents and our implementation can handle this.

Acknowledgements

The authors would like to thank David Parker who pointed out an error in the implementation. The correction of the error led to a great performance improvement. Pierre-Yves Schobbens should also be acknowledged for his contributions to the formulation of RW in [16]. Hasan Qunoo pointed out some typographical errors and made some useful suggestions about the presentation.

Appendix A. Syntax of the RW language

Here we use standard symbols for syntax definition. $(A)*$ means A repeats zero or more than zero times. $[A]$ means A is optional. $A|B$ means a choice between A and B. Characters quoted by “ ” is a string. All the grammatical units are enclosed by $\langle$ and $\rangle$.

```
⟨Model⟩ ::= ⟨Program⟩ [⟨RunStatement⟩] [⟨Specification⟩]
⟨Program⟩ ::= “AccessControlSystem” ⟨ModelName⟩ ⟨Body⟩ “End”
⟨Body⟩ ::= [⟨ClassDefSection⟩] ⟨PredicateDefSection⟩ ⟨Rules⟩
⟨ClassDefSection⟩ ::= “Class” ⟨ClassName⟩ (“,” ⟨ClassName⟩)* “;”
⟨PredicateDefSection⟩ ::= “Predicate” ⟨PredicateDef⟩ (“,” ⟨PredicateDef⟩)* “;”
⟨PredicateDef⟩ ::= ⟨PredicateName⟩ “(” ⟨ParameterName⟩ “:” ⟨ClassName⟩
                        (“,” ⟨ParameterName⟩ “:” ⟨ClassName⟩)* “)” [“!”]
⟨Rules⟩ ::= ⟨Rule⟩ (⟨Rule⟩)*
⟨Rule⟩ ::= ⟨AccessPattern⟩ “{” [⟨ReadStatement⟩] [⟨WriteStatement⟩] “}”
⟨AccessPattern⟩ ::= ⟨PredicateName⟩ “(”⟨FormalParameter⟩ (“,” ⟨FormalParameter⟩)* “)”
⟨ReadStatement⟩ ::= “read” “:” ⟨Formula⟩ “;”
⟨WriteStatement⟩ ::= “write” “:” ⟨Formula⟩ “;”
⟨Formula⟩ ::= “true” | ⟨ConditionalFormula⟩
⟨ConditionalFormula⟩ ::= ⟨ImplicationFormula⟩
⟨ImplicationFormula⟩ ::= ⟨OrFormula⟩ (⟨implies⟩ ⟨OrFormula⟩)*
⟨OrFormula⟩ ::= ⟨AndFormula⟩ (⟨or⟩ ⟨AndFormula⟩)*
⟨AndFormula⟩ ::= ⟨OtherFormula⟩ (⟨and⟩ ⟨OtherFormula⟩)*
⟨OtherFormula⟩ ::= ⟨AtomicFormula⟩ | “(”(⟨ConditionalFormula⟩)*“)”
                          | ⟨negation⟩ ⟨OtherFormula⟩ | ⟨QuantifiedFormula⟩
⟨AtomicFormula⟩ ::= ⟨SinglePredicate⟩ | ⟨EquivalentFormula⟩
⟨SinglePredicate⟩ ::= ⟨PredicateName⟩ “(”⟨FormalParameter⟩ (“,” ⟨FormalParameter⟩)* “)”
⟨EquivalentFormula⟩ ::= ⟨Term⟩ “=” ⟨Term⟩
```

```
⟨Term⟩ ::= ⟨FormalParameter⟩ | ⟨QuantifiedVariable⟩
⟨QuantifiedFormula⟩ ::= “E”|“A” ⟨QuantifiedVariablesDef⟩ (“,” [“E”|“A”]
                        ⟨QuantifiedVariablesDef⟩)* “[”⟨ConditionalFormula⟩“]”
⟨QuantifiedVariablesDef⟩ ::= [“disj”§] ⟨QuantifiedVariable⟩ (“,” ⟨QuantifiedVariable⟩)*
                             “:” ⟨ClassName⟩
⟨ModelName⟩ ::= ⟨Id⟩
⟨ClassName⟩† ::= ⟨Id⟩
⟨PredicateName⟩ ::= ⟨Id⟩
⟨ParameterName⟩‡ ::= ⟨Id⟩
⟨FormalParameter⟩ ::= ⟨Id⟩
⟨QuantifiedVariable⟩ ::= ⟨Id⟩

⟨implies⟩ ::= “implies” | “→”
⟨or⟩ ::= “or” | “|”
⟨and⟩ ::= “and” | “&”
⟨negation⟩ ::= “∼”
⟨Id⟩ ::= ⟨Letter⟩ (⟨Letter⟩ | ⟨Digit⟩ | “_” | “-”)*
```

† ⟨ClassName⟩ must start with a upper case letter.
‡ ⟨ParameterName⟩ must start with a lower case letter.
§ The keyword “disj” can only be used on quantified variables defined in ⟨CheckStatement⟩
The precedence is: “=” > “∼” > “&” > “|” > “→”

```
⟨RunStatement⟩ ::= “run for” ⟨NumberClassPair⟩ (“,” ⟨NumberClassPair⟩)*
⟨NumberClassPair⟩ ::= ⟨Integer⟩ ⟨ClassName⟩

⟨Specification⟩ ::= ⟨CheckStatement⟩
⟨CheckStatement⟩ ::= “check” “{” ⟨QuantifiedVariablesList⟩ “||”
                     [⟨Conditions⟩ “→”] ⟨Coalition⟩ “:” ⟨Goal⟩ “}”
⟨QuantifiedVariablesList⟩ ::= “E”|“A” ⟨QuantifiedVariablesDef⟩
                              (“,” [“E”|“A”] ⟨QuantifiedVariablesDef⟩)*
⟨Conditions⟩ ::= ⟨Condition⟩ (⟨and⟩ ⟨Condition⟩)*
⟨Condition⟩ ::= ⟨PositiveCondition⟩ | ⟨NegativeCondition⟩
⟨PositiveCondition⟩ ::= ⟨PredicateName⟩ “(”⟨FormalParameter⟩
                        (“,” ⟨FormalParameter⟩)*“)” “*” | “!” | “*!”
⟨NegativeCondition⟩ ::= ⟨negation⟩ ⟨PredicateName⟩ “(”⟨FormalParameter⟩
                         (“,” ⟨FormalParameter⟩)*“)” “!” | “*!”
⟨Goal⟩ ::= [“(”] ⟨OrGoal⟩ [⟨SubGoal⟩] [“)”]
⟨SubGoal⟩ ::= “AND” ⟨Coalition⟩ “:” “(”⟨OrGoal⟩ (⟨SubGoal⟩)* “)”
⟨OrGoal⟩ ::= ⟨AndGoal⟩ (⟨or⟩ ⟨AndGoal⟩)*
⟨AndGoal⟩ ::= ⟨AtomicGoal⟩ (⟨and⟩ ⟨AtomicGoal⟩)*
⟨AtomicGoal⟩ ::= ⟨ReadingGoal⟩ | ⟨RealisingGoal⟩ | ⟨MakingGoal⟩ | “(”⟨Goal⟩“)”
⟨ReadingGoal⟩ ::= “[”⟨GoalExpression⟩“]”
⟨RealisingGoal⟩ ::= “<”⟨GoalExpression⟩“>”
⟨MakingGoal⟩ ::= “{”⟨GoalExpression⟩“}”
⟨GoalExpression⟩ ::= ⟨OrGoalExpression⟩ (⟨implies⟩ ⟨OrGoalExpression⟩)*
```

```
⟨OrGoalExpression⟩ ::= ⟨AndGoalExpression⟩ (⟨or⟩ ⟨AndGoalExpression⟩)*
⟨AndGoalExpression⟩ ::= ⟨BasicGoalExpression⟩ (⟨and⟩ ⟨BasicGoalExpression⟩)*
⟨BasicGoalExpression⟩ ::= ⟨AGoalPredicate⟩ | ⟨negative⟩ ⟨BasicGoalExpression⟩
                          | “(”⟨GoalExpression⟩“)”
⟨AGoalPredicate⟩ ::= ⟨SinglePredicate⟩
⟨Coalition⟩ ::= “{” ⟨Id⟩ (“,” ⟨Id⟩)*“}”
```

Appendix B. Algo-1

The differences between Algo-0 and Algo-1 are the ways they treat those newly found sets through the pre-computations. Algo-0 discards a newly found set if all the states in this set have already in states_seen. Algo-1 discards a newly found set if this set is a subset of a set in strategies. Algo-0 adds a pair constructed from a newly found set to strategies no matter $k_{\mathtt{init}}$ is in the set or not. Algo-1 adds a pair constructed from a newly found set to strategies if $k_{\mathtt{init}}$ is not in the set.

When there are no strategies, both Algo-0 and Algo-1 find none. When there are some strategies, both Algo-0 and Algo-1 find some, however, the strategies found by Algo-1 may differ from the ones found by Algo-0. The pseudo-code of Algo-1 is:

```
strategies := $\emptyset$;
put ($K_{\mathtt{G}}$, skip;) in strategies;
repeat until strategies does not change{
    choose $(Y_1, s_1) \in$ strategies;   // for all pairs in strategies
    for each $p \in P${
        for each $a \in A${
            $PTY_1$ := $\mathtt{Pre}^{\exists,a}_{p:=\top}(Y_1)$;
            if (($PTY_1 \neq \emptyset$) $\wedge$ ($\mathbf{PTY_1} \not\subseteq$ any set of the pairs in strategies)){
                $pts_1$ := “set $p$ to $\top$ by $a$;” + $s_1$;
                if ($k_{\mathtt{init}} \in PTY_1$)
                    output $pts_1$;
                else
                    strategies := strategies $\cup$ ($PTY_1$, $pts_1$);
            }
            $PFY_1$ := $\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y_1)$;
            if (($PFY_1 \neq \emptyset$) $\wedge$ ($\mathbf{PFY_1} \not\subseteq$ any set of the pairs in strategies)){
                $pfs_1$ := “set $p$ to $\bot$ by $a$;” + $s_1$;
                if ($k_{\mathtt{init}} \in PFY_1$)
                    output $pfs_1$;
                else
                    strategies := strategies $\cup$ ($PFY_1$, $pfs_1$);
            }
        }
    }
    choose $(Y_2, s_2) \in$ strategies;   // for all pairs in strategies
    for each $p \in P${
        for each $a \in A${
            $PSY$ := $\mathtt{Pre}^{\exists,a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{p=\bot}(Y_2)$;
            if (($PSY \neq \emptyset$) $\wedge$ (**PSY $\not\subseteq$ any set of the pairs in** strategies)){
                $pss$ := "if ($p$) by a then $s1$ else $s2$";
                if ($k_{\mathtt{init}} \in PSY$)
                    output $pss$;
                **else**
                    strategies **:=** strategies $\cup$ $(PSY, pss)$**;**
            }
        }
    }
}
```

Appendix C. Command line parameters

The format of the command which executes AcPeg, is

```
java RWcheck parameters filename [abstraction_level]
```

where parameters is a list of characters which specify the behaviour of AcPeg; filename is the name (path may be included) of the RW script which defines a RW model (and a property); and abstraction_level (optional) is an integer which, in the mode of model-checking, specifies the abstraction level at which AcPeg, runs.

The list of parameters must start with either 'o', which tells AcPeg, to do the translation, or 'c' which tells AcPeg, to do the model-checking. If 'o' is used, AcPeg, will translate the policy defined by the program in the RW script to a XACML policy file and save it in the same directory with the script. The translated XACML file has the same name with the script, except bearing the extension '.xml'. If 'c' is used, other parameters should be specified to further regulate AcPeg's behaviour. We summarise the usage of these parameters in the mode of the model-checking in the following list.

a/i 'a' is for 'searching for strategies' and 'i' is for 'searching for guessing strategies'. 'a' and 'i' cannot both occur in the command line.

p 'p' is for 'outputting strategies or guessing strategies'. If 'p' is used, before each round of a checking starts, AcPeg, prompts a question, asking whether strategies or guessing strategies found in this round should be outputted. If, 'p' is not used, AcPeg, does not prompt the question at the beginning of each round, but only returns an answer ‘yes’ or ‘no’ for this round of checking.

g ‘g’ tells AcPeg, to perform counter-example guided abstraction refinement (CEGAR), in the case that an abstraction level is specified.

r ‘r’ tells AcPeg, to run every round of a checking. Otherwise, AcPeg, prompts a question before the starting of each round, asking whether this round should be running.

0/1 ‘0’ is for ‘running in Algo-0’ and ‘1’ is for ‘running in Algo-1’. Like ‘a’ and ‘c’, ‘0’ and ‘1’ are mutual-exclusive in the command line.

The manual and other documentation can be found in [33].

References

[1] S. Abiteboul, R. Hull and V. Vianu, *Foundations of Databases*, Addison Wesley, 1995, Chapter 7.

[2] T. Ahmed and A.R. Tripathi, Static verification of security requirements in role based CSCW systems, in: *SACMAT’03*, Como, Italy, Jun 2003.

[3] R. Alur, L. deAlfaro, T.A. Henzinger, S.C. Krishnan, F.Y.C. Mang, S. Qadeer, S.K. Rajamani and S. Tasiran, *Mocha User Manual*, The manual and the model checker can be obtained from http://www.eecs.berkeley.edu/~mocha

[4] R. Alur, T.A. Henzinger and O. Kupferman, Alternating-time temporal logic, in: *Compositionality – The Significant Difference, LNCS 1536*, Springer-Verlag, 1999, pp. 23–60.

[5] J. Bacon, K. Moody and W. Yao, A model of OASIS role-based access control and its support for active security, in: *6th ACM symposium on Access control models and technologies*, Chantilly, VA, USA, ACM Press, 2001, pp. 171–181.

[6] E.S. Barka, Framework for role-based delegation models, PhD thesis, George Mason University, 2002.

[7] A. Belokosztolszk and K. Moody, Meta-policies for distributed role-based access control systems, in: *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY’02)*, IEEE Computer Society, 2002, p. 106.

[8] R. Cavada, A. Cimatti, E. Olivetti, G. Keighren, M. Pistore and M. Roveri, *NuSMV 2.2 User Manual*, 2005. This document and the model checker can be obtained from http://nusmv.irst.itc.it

[9] A.K. Chandra and P.M. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: *9th Annual ACM Symposium on Theory of Computing*, Boulder, CO, USA, ACM Press, 1977, pp. 77–90.

[10] B. Chess, Improving computer security using extended static checking, in: *2002 IEEE Symposium on Security and Privacy*, Washington, DC, USA, IEEE Computer Society, 2002.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, Counterexample guided abstraction refinement for symbolic model checking, *Journal of the ACM (JACM)* **50** (2003), 752–794.

[12] S.D.C. di Vimercati, S. Paraboschi and P. Samarati, Access control: principles and solutions, *Software Practice and Experience* **33** (2003), 397–421.

[13] E.A. Emerson, Temporal and modal logic, in: *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, MIT Press, Cambridge, MA, USA, 1991, pp. 995–1072.

[14] K. Fisler, S. Krishnamurthi, L.A. Meyerovich and M.C. Tschantz, Verification and change-impact analysis of access-control policies, in: *ICSE’05*, St. Louis, Missouri, USA, May 2005.

[15] S. Godik and T. Moses, *eXtensible Access Control Markup Language*, OASIS committee, 1.1 edition, Aug 2003, Committee specification.

[16] D.P. Guelev, M.D. Ryan, and P.-Y. Schobbens, Model-checking access control policies, in: *7th Information Security Conference (ISC'04)*, *Lecture Notes in Computer Science*, Springer-Verlag, 2004.

[17] J.D. Guttman and A.L. Herzog, Rigorous automated network security management, *International Journal of Information Security*, Dec 2004.

[18] J.D. Guttman, A.L. Herzog, J.D. Ramsdell and C.W. Skorupka, Verifying information flow goals in security-enhanced Linux, in: *Workshop on Issues in the Theory of Security*, Jan 2004.

[19] J.Y. Halpern and V. Weissman, Using first-order logic to reason about policies, in: *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, Pacific Grove, CA, 2003.

[20] M.R.A. Huth and M.D. Ryan, Verification by model checking, in: *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2nd edition, 2004, pp. 172–255, Chapter 3.

[21] M.R.A. Huth and M.D. Ryan, Binary decision diagrams, in: *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2nd edition, 2004, pp. 358–413, Chapter 6.

[22] D. Jackson, Alloy: A lightweight object modelling notation, in: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM Press, 2002, p. 256–290.

[23] D. Jackson, *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, Software Design Group, MIT Lab for Computer Science, Feb 2002. This document and the tool can be obtained from http://alloy.mit.edu/

[24] A. Jones and M. Sergot, On the characterisation of law and computer systems: The normative systems perspective, in: J. Meyer and R. Wieringa, eds, *Deontic Logic in Computer Science. Normative system Specification*, Wiley, 1993, pp. 275–307.

[25] K.L. McMillan, Symbolic model checking, PhD thesis, School of Computer Science, Carnegie Mellon University, 1993.

[26] R. Sandhu, Separation of duties in computerised information systems, in: *IFIP WG11.3 Workshop on Database Security*, Sep 1990.

[27] R. Sandhu, V. Bhamidipati and Q. Munawer, The ARBAC97 model for role-based administration of roles, *ACM Transactions on Information and System Security* **2**(1) (1999), 105–135.

[28] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, Role-based access control models, *IEEE Computer* **29**(2) (1996), 38–47.

[29] A. Schaad and J. Moffett, A lightweight approach to specification and analysis of role-based access control extensions, in: *SACMAT'02*, Monterey, California, USA, Jun 2002.

[30] Sun Microsystems, Sun's XACML implementation, Aug 2003. Information about this implementation can be found at http://sunxacml.sourceforge.net/

[31] J. Whaley, JavaBDD: Java BDD implementation, 2004. Information about this implementation can be found at http://javabdd.sourceforge.net/

[32] N. Zhang, Verification of access control systems using Mocha, Master's thesis, School of Computer Science, University of Birmingham, 2002.

[33] N. Zhang, AcPeg, the access control policy evaluator and generator, July 2006, The tool can be obtained from www.cs.bham.ac.uk/~nxz or www.cs.bham.ac.uk/~mdr/research/projects/05-AccessControl

[34] N. Zhang, M. Ryan and D.P. Guelev, Synthesising verified access control systems in XACML, in: *2004 ACM Workshop on Formal Methods in Security Engineering*, Washington DC, USA, Oct 2004, ACM Press, pp. 56–65.

[35] N. Zhang, M.D. Ryan and D.P. Guelev, Evaluating access control policies through model-checking, in: *8th Information Security Conference (ISC'05)*, Singapore, Sep 2005, Springer-Verlag, 2005.