Generating Verified Access Control Policies through Model-Checking

by

Nan Zhang

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
October 2005
The glory of the LORD shall be revealed, and all flesh shall see it together.

Isaiah 40:5
Acknowledgements

It is hard to express in words that how much I am indebted to my supervisor, Mark Ryan, for without his help and guidance this work could have never been finished. In fact, if it was not because of his wisdom and insight this work would have never started. From him I have learnt a lot, especially on how to be earnest and rigorous in research. He is an authentic example of a devoted scientist as well as a patient, considerate, friendly and wise supervisor. In the time of my frustration, he encouraged me. In the time of my pain, he comforted me. In the time of my confusion, he enlightened me. In the time of my worry, he assured me. Always, he gave me timely assistance during our weekly discussions. Whenever I got stuck, I knew that he is always there. Perhaps, you can imagine how joyful I was when, many times, a tricky problem was solved following his advice. I really believe that he is a blessing that God has given me.

Another blessing that God has given me is my dear colleague, Dimitar P. Guelev, perhaps the cleverest person I have ever seen on this planet. I could hardly think of a more suitable word to describe him than ‘genius’. Had he not made many invaluable suggestions and even challenges, this work would have never reached the current stage.

My sincere gratitude should also go to Marta Kwiatkowska and Eike Ritter. Being members of my thesis group, they are acknowledged for their time, effort and guidance during my PhD research.

The efforts of Ken Moody spent on reading my thesis should also be acknowledged. Being my external examiner, he pointed out a large number of misprints that were in my thesis. From the corrections he suggested, I learnt grammatical rules that I never noticed before.

Last but not least I must thank David Parker who pointed out a mistake in my implementation. The performance of the tool was greatly improved after the correction.
Abstract

This thesis presents a framework for evaluating and generating access control policies. The framework contains a modelling formalism (RW formalism), a machine-readable language (RW language), a model-checking algorithm and a tool (AcPeg) implemented in Java. The RW formalism is for modelling access control policies. (The term ‘access control policy’ here refers to a set of authorisation rules.) The RW language is for expressing policies modelled in the RW formalism and security properties to be verified. A property defines a goal, a coalition of agents and some conditions. A goal includes 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. The conditions defined by the property serve as pre-requirements based on which the checking is performed. In the case that the goal is achievable, the algorithm outputs strategies that 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, among which are security holes caused by interaction of rules, co-operation between agents (including changing of each other’s privileges) and multi-step actions. When malicious goals are achievable, the outputting 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. A real access control system can then be built on the converted policy file.
Contents

1 Introduction

1.1 Thesis problem ......................................................... 1
1.2 Why previous work is not adequate ............................... 2
1.3 Our solution .......................................................... 4
1.4 General description of the RW model-checking algorithm ....... 7
1.5 Contributions of the thesis ........................................... 7
1.6 Structure of the thesis ................................................ 8
1.7 Other publications ...................................................... 8

2 Background material and related work ................................. 9

2.1 Outline ................................................................. 9
2.2 Access control ......................................................... 10
  2.2.1 Authentication and authorisation ............................... 10
  2.2.2 Access control policy ............................................ 10
  2.2.3 Models of access control policy ................................. 11
    2.2.3.1 Access control matrix .................................. 12
    2.2.3.2 Mandatory access control ............................... 14
    2.2.3.3 Discretionary access control ............................ 15
    2.2.3.4 Role-based access control ............................... 16
  2.2.4 Languages used for specifying access control policy ......... 20
    2.2.4.1 XACML ...................................................... 20
    2.2.4.2 RDL ......................................................... 24
    2.2.4.3 Others ....................................................... 20
  2.3 Model-based verification .............................................. 33
    2.3.1 Model-checking ................................................ 33
5 RW model-checking
5.1 Outline .......................................................... 63
5.2 The RW model-checking problem .................................. 63
5.3 Assumptions made by the algorithm .................................. 63
5.4 The transition system ................................................. 63
5.5 Representation of $k_{init}$ ........................................... 63
5.6 Representation of $K_0$ ............................................... 63
  5.6.1 Substitution of reading goals .................................... 63
  5.6.2 Substitution of realising goals ................................... 63
  5.6.3 Substitution of making goals ................................... 63
5.7 Backwards reachability computation .................................. 63
  5.7.1 Computing sets of states ...................................... 63
  5.7.2 Generating strategies ......................................... 63
5.8 Pseudo-code for the algorithm ...................................... 63
5.9 Proof of correctness ................................................ 63
5.10 Computational complexity ......................................... 63

6 Implementation
6.1 Outline .......................................................... 77
6.2 General description ............................................... 77
6.3 Computational round .............................................. 77
6.4 Strategies and guessing strategies ................................. 77
6.5 Abstraction ....................................................... 77
6.6 Strategies for nested queries ....................................... 77
6.7 Case study – an employee information system ...................... 77
6.8 Case study – a student information system ....................... 77
6.9 Case study – a patient record system .............................. 77
6.10 Performance ........................................................ 77

7 RW to XACML
7.1 Outline .......................................................... 92
7.2 Compiling RW to XACML and SQL ................................. 92
  7.2.1 Structure of the converted XACML file ....................... 92
  7.2.2 Generating the condition tag .................................. 92
CONTENTS

7.2.3 The external function – self-defined: evaluate-sql .......................... 98
7.3 Assumptions about the database ................................................. 98

8 Discussions and conclusions ....................................................... 101
  8.1 Summary ............................................................... 101
  8.2 Practical applicability of the RW framework ............................... 102
      8.2.1 The modelling power of the RW formalism .......................... 102
      8.2.2 The expressive power of the RW language .......................... 102
      8.2.3 The checking ability of AcPeg ................................... 103
  8.3 General steps of using the RW framework to verify security properties 103
  8.4 Amending policies ...................................................... 104
  8.5 On the translation to XACML .............................................. 105
  8.6 Future work ............................................................ 106

A Syntax of the RW language ....................................................... 107

B Algo-1 ............................................................................. 109

C Command line parameters ......................................................... 111

Bibliography ............................................................................ 113
# List of Figures

2.1 An access control matrix .................................................. 13
2.2 An access control list .................................................... 13
2.3 The RBAC0 model ........................................................... 16
2.4 A role-hierarchy .............................................................. 17
2.5 The relation between the four models in the RBAC model family ... 17
2.6 The RBAC3 model ............................................................ 18
2.7 The data-flow model of XACML ......................................... 21
2.8 An example for the head and the description of a rule in XACML . 22
2.9 An example for the target in XACML .................................. 23
2.10 An example for the condition in XACML .............................. 23
2.11 A request in XACML ....................................................... 24
2.12 The authorisation of an OASIS service ............................... 26
2.13 The interworking between OASIS services ........................... 27
2.14 Authorisation in trust-management systems ........................... 30
2.15 A SMV program and a specification .................................. 32
2.16 Two possible outcomes of a sampling ................................ 36
2.17 A REACTIVEMODULES program ..................................... 38
2.18 Evolution from a binary decision tree to a binary decision diagram. 40
2.19 A simple transition system .............................................. 40
2.20 The BDD representation for the transition relation of the system in Figure 4 19 42
2.21 An Alloy representation and a specification .......................... 44
2.22 A MTBDD for a simple policy ......................................... 47

3.1 The RW script for the policy defined in Example 3.1 .................. 55

5.1 The transitions ............................................................... 66
### LIST OF FIGURES

5.2 The process of learning .................................................. 56

6.1 The RW script for the policy defined in Example 6.1 .......................... 76
6.2 The guessing strategy found for Example 6.2 .............................. 81
6.3 A false strategy and a suggestion made by AcPeg when checking Query 4.2 in abstraction level 2 and CEGAR .................................................. 82
6.4 The strategy found for Query 4.3 ........................................ 83
6.5 Two strategies found for Query 6.2 ....................................... 84
6.6 The strategy found for Query 4.1 ........................................ 85
6.7 The RW script for the access control system in Example 6.2 .................. 86
6.8 A strategy found for Query 6.3 ........................................ 87
6.9 The RW script for the access control system in Example 6.3 .................. 88
6.10 The RW script for the access control system in Example 6.3 ................. 89

7.1 The structure of an output XACML policy file .............................. 95
7.2 Correspondence between a privilege definition and its generated rule ........... 94
7.3 A request for the rule in Figure 7.2 ..................................... 95
7.4 A formula in a RW script and its generated Condition tag ...................... 96
7.5 Tables and the structure of the database set up for the conference example ... 96

8.1 The amended version of rule 1.1 in Example 8.1 ............................ 104
8.2 The amended version of rule 4.1 in Example 8.1 ............................ 105
8.3 The accessing rules for the predicate assigned(paper : Paper, agent : Agent) ! 105
List of Tables

2.1 The truth table for \( \neg(x \lor y) \) .................................................. 38
2.2 Representing subsets of states by propositional formulas. ....................... 41
2.3 Differences between Alloy and model-checking .................................... 45

6.1 Mappings \( x(p,a) \) and \( w(p,a) \) for the access control system in Example 6.1 75
6.2 Experimental results obtained from running the test cases in Chapter 6 ........ 91
6.3 Experimental results obtained from checking Query 6.3 with different sizes of the model. ........................................ 93
Nomenclature

Acronyms

**AcPeg**  Access Control Policy Evaluator and Generator

**ACL**  Access Control List

**ARBAC97**  Administrative Role-Based Access Control 97

**ATL**  Alternating-Time temporal Logic

**BDDs**  Binary Decision Diagrams

**CEA**  Cambridge Event Architecture

**CEGAR**  CounterExample-Guided Abstraction Refinement

**CNF**  Conjunctive Normal Form

**CTL**  Computation Tree Logic

**D1LP**  version 1 of Delegation Logic Programs

**DAC**  Discretionary Access Control

**DL**  Delegation Logic

**DNF**  Disjunctive Normal Form

**EHRs**  Electronic Health Records

**EPLA**  Enterprise Privacy Authorisation Language

**LTL**  Linear-Time temporal Logic

**MAC**  Mandatory Access Control
NOMENCLATURE

MTBDDs Multi-Terminal Binary Decision Diagrams

NHS National Health Service

NuSMV New Symbolic Model Verifier

OASIS Organisation for the Advancement of Structured Information Standards; or Open Architecture for Security Interworking Services

PAP Policy Administration Point

PDP Policy Decision Point

PEP Policy Enforcement Point

PIP Policy Information Point

RBAC Role-Based Access Control

RBDM0 Role-Based Delegation Model 0

RBDM1 Role-Based Delegation Model 1

RCL2000 Role-Based Constraints Language 2000

RDL Role Definition Language

RMC Role Membership Certificate

RSL99 Role-Based Separation of Duty Language 1999

RW Read and Write; also the name given to the system we present

SELinux Security-Enhanced Linux

SMV Symbolic Model Verifier

TM Trust-Management

XACML eXtensible Access Control Markup Language

Definitions of Terms

Access control list A column of an access control matrix which specifies privileges that all subjects have on the object designated by the column.
**Access control matrix** A table with rows named by subjects and columns named by objects, and each cell in the table defines privileges that the subject of the row have upon the object of the column.

**Access control policy model** A formal template which contains a set of strategies for defining access control policies.

**Access control policy** Pre-defined rules used in an access control system to regulate the behaviours of the system when facing accessing requests.

**Access control** The process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied.

**Authentication** The process of deciding the identity of a subject.

**Authorisation** The process of deciding whether a request should be granted or denied based on the outcome of authentication.

**Capability list** A row of an access control matrix which defines total privileges of the subject designated by the row.

**Closed system** A system whose behaviour is completely determined by the state of the system.

**Concurrent system** A system in which several processes run simultaneously.

**Delegation** The mechanism, in access control systems, that allows an agent to give part or all of his privileges to another agent so that the latter can carry out actions on behalf of the former.

**Open system** A system which interacts with its environment and whose behaviour depends on the state of the system as well as the behaviour of the environment.

**Permission about permissions** Permissions, in access control systems, that can give agents abilities to change other agents’ permissions.

**Proof-of-compliance question** The question of whether a set of credentials carried with a request compiles with a policy.

**Reactive system** A system which reacts to its environment and is not meant to terminate.
Trust management systems Systems of managing authorisation in large-scale, open and distributed environments, where authorisation decisions cannot base on the identities of requesters but on the proof-of-compliance approach.
Chapter 1

Introduction

1.1 Thesis problem

The ever-more interconnection of computers through networks makes resource-sharing much more convenient than before. However, this universal sharing of resources has also caused big security challenges, especially, on how to guarantee that information is accessed in legitimate ways. To meet this need, access control is widely used in systems using computers. Just to name a few examples of these systems: Access controls used in operating systems in banking systems [And01], in on-line communities [Zha02], and in health care systems [LLTC00, And96]. These systems can provide satisfactory services to their users only if the enforced access control mechanisms are reliable.

Access control mechanisms enforced by access control systems are usually built on access control policies. An access control policy normally consists of a number of authorisation rules which specify the abilities of the users to access their respective kinds of resources and prevent access that can serve identified adverse purposes. The reliability of an access control mechanism, therefore, largely depends on the fitness of the adopted policy. An access control policy being fit means it should provide enough permissions for its users to carry out legitimate actions and prevent them from accessing information in illegitimate ways. Given an access control policy, from the perspective of security analysis we may want to evaluate whether it is fit for the services it intends to provide. If, in some aspect, it is unfit, we may need to know how it is unfit and therefore find ways to amend it. From the perspective of implementation, it is extremely useful if reliable policies can be generated automatically in an access control policy specification language used in practice. The research work presented in this thesis aims at
providing solutions to meet these needs.

1.2 Why previous work is not adequate

Using tools to evaluate access control policies automatically is much preferable to evaluating them by hand, because the complexity of access control policies often makes reasoning about them by hand infeasible. Researchers have made use of program tools in the analysis of access control policies, some of which have been developed for this purpose. To list just a few examples of these approaches: Fisker, Krishnamurthi, Meyerovich and Tschantz\(^\text{[FMT05]}\) create Margrave, a software suite for analysing role-based access control policies (RBAC\(^\text{[SCF96]}\)). Margrave reads policies in XACML (eXtensible Access Control Markup Language\(^\text{[GM03]}\)), translating them into multi-terminal decision diagrams (MTBDDs\(^\text{[CFM+93]}\)). With the help of these MTBDDs, Margrave verifies whether a rule preserves a property by taking a query that expresses the property as input and returning an answer to the query. Halpern and Weissman\(^\text{[HW03]}\) demonstrate how a fragment of first-order logic can be used to represent and reason about access control policies. A prototype that has a user graphic interface and a reasoning engine is implemented. Through theorem reasoning the engine answers whether a request should be granted based on certain policy statements. Schaad and Moffett\(^\text{[SM02]}\) use Alloy\(^\text{[Jac02],[Jac02b]}\) to specify a RBAC-style model\(^\text{[SCF96]}\), ARBAC97-style extensions\(^\text{[SBM99]}\) and a set of separation of duty properties\(^\text{[San90]}\). They analyse possible conflicts that could arise out of decentralised administrative actions with respect to the separation of duty constraints. Their work demonstrates the general suitability of the Alloy language for specification and analysis of role based systems.

However, what have long been neglected by much previous work are the analysis and the detection of security holes in policies caused by \textbf{interactions of rules, co-operations between agents} and \textbf{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;
1.2 Why previous work is not adequate

- 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 \( 4 \) 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 \( 4 \):

**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 being reviewer for \( p \). (permitted by rule \( 1 \))
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, both strategies enable \( a \) and \( c \) to by-pass rule \( 4 \). Three reasons have caused this problem:
1. **Interactions of rules.** Although rule $\text{3}$ explicitly prohibits a reviewer from reading another reviewer’s review for the same paper assigned to both of them, rule $\text{2}$ and rule $\text{4}$ provide opportunities for the agents to by-pass it.

2. **Co-operations between agents.** Although $a$ cannot breach rule $\text{3}$ by himself, with the help of $c$, they can act together through an indirect way to get around rule $\text{3}$. 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 $\text{2}$ 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 $\text{[AT03, Ch02]}$ and policy-querying $\text{[FKMT05]}$.

### 1.3 Our solution

The framework presented in this thesis 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 $\text{RW}$ 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.
1.3 Our solution

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 [BM02]). 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 [McM93]. 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 how to amend the policy.

AcPeg. This is a tool written in Java, which implements the above algorithm to perform the checking. Moreover, it can translate the policy-description in the RW language into a policy file in XACML [GM02]. 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.
1.4 General description of the RW model-checking algorithm

The core of our work is the RW model-checking algorithm and its implementation.

Given an access control system $S$, a set of agents $A$, and a goal $G$ that $A$ wants to achieve within the permissions $S$ provides, the algorithm figures out whether there are strategies available for $A$ to achieve $G$. A strategy is a sequence of legitimate reading and overwriting steps, each of which is performed by an agent in $A$. Goal $G$ includes reading and overwriting information in $S$. If strategies for $A$ to achieve $G$ exist, the algorithm outputs at least one of them. 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. Therefore, the algorithm assumes that the policy adopted by $S$ is completely known to all the agents.

The algorithm searches for a strategy by considering $A$'s knowledge on the state of $S$ while $A$ carries out the strategy. We use $K_0$ to denote the set of $A$’s knowledge states from each of which $A$ can deduce that $G$ is reached; and $k_{1st}$ to denote $A$’s knowledge state on $S$ at the start of carrying out the strategy. Therefore, a strategy, consisting of actions that cause $A$ to learn and change the state of $S$ (reading actions correspond to learning and overwriting actions correspond to changing), will eventually lead $A$’s knowledge state from $k_{1st}$ to a state in $K_0$. This is so because if an agent $a \in A$ overwrites a piece of information $p$ in $S$, $A$ knows the current value of $p$ and at the same time changes the current state of $S$. The knowledge of $p$’s value contributes $A$’s knowledge on the current state of $S$. If $a$ reads $p$ which has not been overwritten, $A$ learns the initial value of $p$, that is, $p$’s value at the starting of executing the strategy. This knowledge contributes $A$’s knowledge on the initial state as well as the current state of $S$. Consequently, these reading and overwriting actions cause $A$’s knowledge on $S$ transit from one state to another. The model-checking algorithm ranges over the transition system built upon $A$’s knowledge states and searches for strategies that lead $A$’s knowledge from $k_{1st}$ to $K_0$ by backwards reachability.

1.5 Contributions of the thesis

The framework presented in this thesis contains work from my colleagues. A mathematical formulation of RW was first proposed in [GR0], and a Prolog-based procedure for deciding the achievability of a goal was presented, without outputting strategies. The work presented in
this thesis can be considered as an extension and continuation of the work described in \cite{GRS04}. The contributions of this thesis are:

- A better understanding of the security vulnerabilities in access control policies caused by interactions of rules, co-operation between agents (including changing of each other’s privileges) and multi-step actions. A solution for detecting and amending these vulnerabilities is presented.

- A machine-readable syntax for the \textit{RW} language, both for policy description and property specification, is designed.

- The \textit{RW} model-checking algorithm is presented. In contrast with the inefficient prototype Prolog rules of \cite{GRS04}, the model-checking algorithm here is based on a transition system of knowledge states. It is implemented using BDDs.

- The correctness of the algorithm is proved. Its computational complexity is discussed.

- A solution of generating verified access control policies in XACML is realised by translating policies written in the \textit{RW} language to policies in XACML.

- AcPeg, which does both the model-checking and the translation, is implemented in Java. To increase the performance of the tool, three abstraction levels are integrated into the tool. A mechanism to perform CEGAR (counterexample-guided abstraction refinement \cite{CGJ+03}) is also implemented.

- The usability of the tool is demonstrated through several case studies. Experimental results are discussed. The general usability of model-checking in the area of access control policy analysis is demonstrated.

1.6 Structure of the thesis

Background material and related work are discussed in Chapter \ref{chap:background}. The formal definition of the \textit{RW} formalism is introduced in Chapter \ref{chap:rw} as well as the description part of the \textit{RW} language. Writing properties in the \textit{RW} language is explained in Chapter \ref{chap:writing}. The \textit{RW} model-checking algorithm is presented in Chapter \ref{chap:model-checking}. The implementation of the algorithm and experimental results are discussed in Chapter \ref{chap:implementation}. The translation from \textit{RW} to XACML is explained in Chapter \ref{chap:translation}. Conclusions are drawn in Chapter \ref{chap:conclusions}.
1.7 Other publications

Our work has two threads: the RW model-checking and the translation from RW to XACML. The work on the translation has been published in [ZRG04]. The work on the RW model-checking has been published in [ZRG05a]. Another paper titled *Synthesising Verified Access Control Systems through Model-Checking* [ZRG05b] has been submitted to the *Journal of Computer Security*. The content of the paper includes both the RW model-checking and the translation.
Chapter 2

Background material and related work

2.1 Outline

Since the work presented in this thesis relates to both access control and model-based verification, the review presented in the rest of this chapter includes literature in both of the fields.

Literature on access control is discussed in 2.2. The emphasis is on access control policy. Before the discussion on access control policy, the meanings of authentication and authorisation are discussed in 2.2.1. The concept of access control policy is clarified in 2.2.2. Four major models for formulating access control policy are discussed and compared in terms of how they deal with administration, delegation and constraint in 2.2.3. Several formal access control policy specification languages are discussed in 2.2.4 as well as the influences that some of them have on our work.

Literature on model-based verification is discussed in 2.3. It is classified as model-checking (2.3.1) or model-finding (2.3.2). The emphasis is on model-checking and its symbolic approach (2.3.3). In model-checking, CTL-SMV and ATL-Mocha are discussed. In model-finding, Alloy is discussed as well as a comparison with model-checking. In each discussion on CTL-SMV, ATL-Mocha and Alloy, we give some reasons for which they are not suitable for our work. Abstraction is discussed in 2.3.1.4.

Apart from the above, literature on reasoning about access control is discussed in 2.4. Several typical papers in this area are picked. We generally describe the work presented in each paper. When it is appropriate we explain how their work relates to ours or give a comparison between
their work and ours.

2.2 Access control

2.2.1 Authentication and authorisation

‘Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied.’ – [VPS03]. This process usually has two stages – the stage of deciding the identity of the subject who made the request, and then the stage of deciding whether the request should be granted or denied. The first stage is usually known as authentication and the second authorisation.

In centralised environments, where a subject is known by the system, authentication is usually based on user names and passwords. In distributed environments, where a subject is not likely to be known by the system, authentication is usually based on credentials carried with the request [LGF03]. After a subject has been authenticated, the system still needs mechanisms to control which data it can read or write. This is the task of authorisation. Based on the outcome of authentication, authorisation determines whether a request issued by the subject should be granted or denied.

Comment This classification makes it clear to us that we are working in the area of authorisation. In this thesis, the two terms access control and authorisation sometimes are used interchangeably.

2.2.2 Access control policy

A policy is a rule that defines a choice in the behaviour of a system when facing a particular situation [DDL01, Dau02]. An access control policy is a set of pre-defined rules used in an access control system to regulate how the system should behave when facing access requests. The choices involved are deciding whether the requests should be granted or denied. Access control policies are usually declarative, stated through appropriate specification languages and then enforced by the access control mechanisms provided by the system. Separating the policy from the implementation of a system has several advantages, of which the most obvious one is that the policy may be modified in order to dynamically change the behaviour of a system, without changing its underlying implementation [Spo94].

Comment In this thesis, the term ‘access control policy’ refers to a set of authorisation rules.
2.2.3 Models of access control policy

Access control policies are often complex because of the nature of the policies used in real-world situations. In order to approach a better and systematic understanding, formulating and analysing of access control policies, during past years, researchers have proposed several access control policy models. Some of them are still widely used. A model of access control policy is a formal template which contains a set of strategies for how access control policies should be defined. It is not bound to a particular situation or organisation, but must be able to be used in generic ways. As Anderson comments in [And01], a model is often the basis of formal analysis.

Throughout the rest of this sub-section, we will discuss four typical models of access control policy. However before doing that, we shall first discuss three concepts that are important to any model. They are: administrative policies, constraints and mechanisms of delegation.

Administrative policies are policies defining who can add, delete or modify regular policies. Some researchers call them meta-policies [BM02], or as we call them permissions about permissions. The careful definition of administrative policies is important to any access control system. For if they were not carefully defined the whole system could lose the force of protection. Thus any model of access control policy must address the issue of how administrative privileges are organised. There are four major types of solutions [VPS03]:

- **centralised**, where a user or a group of privileged users, known as administrator(s), retain the privileges of granting or revoking permissions;
- **hierarchical**, where the administrative power is distributed among a set of authorised users;
- **ownership**, where the owner of an object can grant or revoke from other users the permissions for accessing her objects;
- **decentralised**, where the administrative power is further distributed among common users through delegation.

Over-centralised solutions should be avoided by any model because they may cause the root-bottleneck problem as well as the misuse of the administrative power. However,

\footnote{Normally, there are hierarchies among this set of users according to how much power each one has.}
2.2.3 Models of access control policy

over-decentralised solutions should also be avoided because they may complicate the authorisation management. Users may find that it is difficult to keep track of who can access their objects.

**Constraints** are certain guidelines that policy-designers should follow and thus the policies should preserve. One of the most famous constraints often enforced in companies is the rule of *separation of duty*. It refers to the principle that no user should be given enough privileges to misuse the system on her own [San90]. A common example in a company is that the position of purchasing manager and accounts payable manager should not be held by the same person.

The challenge that constraints pose on any model is whether the model is expressive enough to express different kinds of constraints. The challenge for researchers who want to analyse access control policies is whether the behaviours of the policies breach certain constraints [AT03, Kol03, SM02].

**Delegation** is the ability of one subject to give all or part of their privileges to another so that the latter can carry out actions on behalf of the former. It is probably the most widely studied topic in the study of security policy [BS00, Bar02, BDF02, LGF03, NC06, NFRB03]. Used properly, delegation may increase the flexibility of an access control system. Used improperly, delegation may cause problems. Through delegation, privileges may be passed to users who should not have them. That is why researchers incline to restrict behaviours of delegation so that they can be constrained into a manageable manner [Bar02, BDF02, NFRB03].

Proper use of delegation makes a system more flexible because it enables an agent to perform another agent’s task in the cases that the latter is not available. It also makes resource-sharing more convenient. For example, the owner of a file may make the file available for others to read by delegating the privilege of reading it to whoever she wants. Furthermore, through delegation, the administrative power is distributed so that the root-bottleneck problem may be relieved.

However, the using of delegation may bring some complicated issues, such as the issue of revocation. This issue is made much more complicated if multi-level delegation is allowed. Barka, in [Bar02], proposes several well-defined delegation models in the context of RBAC to deal with this issue.
2.2.3 Models of access control policy

Delegation may also cause problems in aspects of security. Through delegation privileges may be granted to agents who should not have them. Moreover, delegation may breach certain constraints. Thus, for models including mechanisms of delegation, it is important to ensure that the behaviours of delegation are compatible with other rules and constraints.

Now, we shall turn to the discussion of four particular access control policy models – access control matrix, mandatory access control, discretionary access control, and role-based access control. We shall see how each one of them addresses the three issues mentioned above.

2.2.3.1 Access control matrix

In [HRU76], Harrison, Ruzzo, Ullman present a model of protection mechanisms in computing systems, which is later known as the access control matrix.

In that model, access permissions are represented by a matrix, with columns named by objects and rows named by subjects. Each cell in the matrix contains the operations that the subject of the row can perform onto the object of the column. See Figure 2.1 for an illustration.

```
<table>
<thead>
<tr>
<th></th>
<th>Object b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Subject a</td>
<td>read, write</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Figure 2.1: An access control matrix

Following this model, an access control system can be represented by a triple \((S, O, P)\), where \(S\) is the set of current subjects, \(O\) is the set of current objects, \(S \subseteq O\), and \(P\) is an access matrix. For a \(s \in S\) and \(o \in O\), \(P[s, o]\) defines the operations that \(s\) can perform onto \(o\). The current values of \(S\), \(O\) and \(P\) together define the current state of the system. Through operations one state of the system may transit to another. Within this context, a safety problem is discussed in their paper, and the undecidability of the problem in general situations is proved.

The safety problem is: Given a particular protection system and a generic right \(r\), whether the initial state of the system \(Q_0\) is unsafe for \(r\). \(Q_0\) is unsafe for \(r\), if there is another state \(Q\) and a command \(\alpha\) such that \(Q_0\) can transit to \(Q\) through finite steps of operations and \(\alpha\) leaks \(r\) from \(Q\). \(\alpha\) leaks \(r\) from \(Q\) if \(\alpha\), when run on \(Q\), can execute a primitive operation which enters \(r\) into a cell of \(P\) which did not contain \(r\).
2.2.3 Models of access control policy

In their paper, they do not explicitly mention the issues of administration, constraint and delegation. However, they have something similar to the delegation we are talking about today – ownership privilege. Owners of objects can give reading, writing and executing privileges on their objects to other subjects. Through this way, administrative power is decentralised.

Nowadays, however, as organisations grow, the explicit using of access control matrices becomes very cumbersome. There are three main approaches to this problem: using access control policy languages to state rules defined in the matrices; using groups and roles to manage the privileges of large sets of users simultaneously; and storing the matrices either by rows (capability lists) or columns (access control lists).

2.2.3.2 Mandatory access control

A system using mandatory access control (MAC) usually enforces security policies independently of user actions [And04]. MAC secures information by assigning a sensitivity level to all information and a sensitivity level to each user. When a user wants to access some information, the system makes a decision by comparing the level of the user to the level of the information. Common features of systems using MAC include [CEH+]:

- All data is assigned a security level that reflects its relative sensitivity.
- Only administrators, not data owners, modify a resource’s sensitivity level.
- All users can read information with classifications no higher that the one they are granted. This is also known as no read up.
- No user can write to a lower classification. This is also known as no write down.

These systems are also known as multilevel security systems. The Bell-LaPadula model [LHM83] is a classical approach for these systems.

MAC is more secure than other models, however, less convenient for users. For this reason, it is often used at military information systems where security is the most important concern. In MAC, administration is centralised. The administrative power is not delegated to common users. Delegation is forbidden because only administrators can modify security levels and thus change users’ privileges. Accesses are extremely constrained in this model.
2.2.3 Models of access control policy

2.2.3.3 Discretionary access control

Unlike MAC, where permissions of a subject cannot be passed to another, discretionary access control (DAC) allows the owner of a permission to pass it to any other subject in the system [SS94]. DAC restricts access to objects based on the identity and need-to-know of users and/or groups to which the object belongs [Har00]. It is widely used in operating systems, such as Windows, Unix and Linux, to restrict users’ access on files.

Access control list (ACL), as a typical DAC approach, is widely implemented in operating systems. An ACL is a column of an access control matrix that specifies permissions on the object of the row for all subjects. In Unix, an ACL is attached to each file and directory. Permissions to the file or directory specified on the list are read (r), write (w) and execute (x) for the owner, the group and others respectively. The list also stores the owner’s identity and the identity of the group that the file belongs to, plus a bit showing whether the object is a file or a directory. Figure 2.2 shows the ACL of a file which belongs to Alice, who is a member of the group Student.

```
-rwxr-x------ Alice  Student
```

Figure 2.2: An access control list.

The first bit on the list shows that the object is a file, not a directory. The following bits specify that the owner, Alice, can read, write and execute the file; members in the group Student can only read the file; and others have no access to the file at all. In Unix each running process bears a user identifier (UID) and a group identifier (GID). The UID is the identity of the user who invokes the process. The GID is the identity of the group that the user belongs to. When a process intends to perform an operation on the file, the system first determines the class that the user, invoking the process, belongs to by comparing the UID and GID of the process with the owner and group information specified on the ACL of the file. After finding out the appropriate class that the user belongs to, the system looks at the permissions specified on the ACL for that class, whether owner, group or others, and makes a judgement whether the intended operation should be allowed.

As the owner of the file, Alice can make the file available for others to read, write and execute by setting corresponding bits using the `chmod` command. Through this way, permissions are delegated to others, however, no further delegation is allowed, because ownership cannot be transformed.
2.2.3 Models of access control policy

The ability to enforce constraints in DAC largely depends on the natures of the constraints. Administration is partly done through delegation, however, a super user, known as the root, ultimately remains control of the system.

2.2.3.4 Role-based access control

The role-based access control (RBAC) model has been proposed more recently as a compromise between MAC and DAC, in the sense that, in RBAC, users can pass their permissions to others, however, they must do so in regulated ways. The central idea of RBAC is that access rights are assigned to agents on the grounds of their having certain roles.

Roles are created for various job functions and users are appointed to roles based on their tasks, qualifications and other reasons. Permissions are associated with roles. User-role relation is a many-to-many relation because a user can have more than one role and a role can be assigned to more than one user. All the roles owned by a user may be activated simultaneously. However, more often, they are activated at different sessions. A user can establish a session during which only a subset of his roles are activated. Thus, a session is a mapping that maps a user to a number of roles. Role-permission relation is also a many-to-many relation as the user-role relation. The idea is captured by Figure 2.3.

![Figure 2.3: The RBAC0 model.](image)

The model described in Figure 2.3 is called RBAC0 in SCFY96. On the basis of RBAC0, more sophisticated models were developed SCFY96. The work goes through two directions. One direction ends up with the RBAC1 model, which has role-hierarchies. A role-hierarchy is a hierarchical structure among roles, which can be expressed by a diagram with vertexes representing roles and edges representing relations. In such a diagram, if there is an edge between two roles, the role on the upper is said to be senior to the role below. Users in a senior role implicitly inherit all the privileges of users in the junior roles, but not vice versa.
Role-hierarchies naturally reflect the hierarchical structures among employees in organisations. For an example of a role-hierarchy, see Figure 2.4

![Figure 2.4: A role-hierarchy](image)

The other line leads to the RBAC2 model, which integrates constraints. At length, both threads end up at the same point which is the RBAC3 model. RBAC3 model integrates both role-hierarchies and constraints. The relation between RBAC0, RBAC1, RBAC2 and RBAC3 is shown in Figure 2.5

![Figure 2.5: The relation between the four models in the RBAC model family.](image)

Figure 2.6 shows the idea of RBAC3 where constraints have effects on UA, PA and RH relations.

Administration in RBAC is discussed by Sandhu, Bhamidipati and Munawer [SBM99]. Administrative tasks in RBAC include assigning users to roles, assigning permissions to roles, enforcing constraints, changing the structures of role-hierarchies, and so on. The simplest assumption is that all these tasks are assigned to a single administrator although users can do part of the user-role assignment by themselves through delegation. However this is unrealistic in practice because the job could be too heavy for a single person. For this reason, a role-based model for RBAC administration is proposed in [SBM99], and it is called the ARBAC97 (Administrative Role-Based Access Control 1997) model.
The basic idea of ARBAC97 is using RBAC to manage RBAC. In this model, administrative tasks are performed by a group of administrative officers, who are separated from common users. These security officers are assigned administrative roles, which, by themselves, form an administrative role-hierarchy. The administrative tasks are distributed among these security officers according to certain policies. The administration of administrative roles and permissions is under the control of the chief security officer.

Delegation between users in RBAC has to do with delegating roles, because in RBAC permissions are associated with roles. Passing permissions from one user to another is done primarily through the changing of users’ roles. Delegation is discussed in BS00, Bar02. The simplest delegation model in RBAC is the RBDM0 (Role-Based Delegation Model 0) model. The assumptions of RBDM0 are:

- Delegation can only happen between a user in a senior role and a user in a junior role. Delegation between users in the same role is not allowed.
- The delegated user cannot do further delegation (one step delegation).
- The delegating user must delegate all his privileges to the delegated user or does not delegate at all (total delegation).

Under these assumptions, delegation defined in the RBDM0 model is: An original member of a senior role can assign a member in a junior role to be a member of the senior role. Any
original member in that senior role can revoke the membership of any delegated member in that role.

However, these assumptions seem to be too restricted for the need of real practice. Other more advanced delegation models are discussed in Bar02. For example, RBDM1 allows partial delegation where a user in a senior role can delegate part of her permissions to a user in a junior role by delegating a sub-role.

Constraint is an important aspect of RBAC and is sometimes argued to be the principal motivation behind RBAC SCFY99. Because in RBAC roles are central links between users and permissions, most constraints apply to user-role relations and role-permission relations, however, sometimes, also to role-role relations. The most common constraints in RBAC are mutually exclusive roles, cardinality and prerequisite roles.

Mutually exclusive roles are a set of roles that cannot be held by the same user at the same time. This supports separation of duties. This constraint on user-role relations also has a counterpart on role-permission relations, which requires that mutually exclusive permissions cannot be assigned to the same role.

Cardinality is a constraint on the maximum number of members in a role. For example, only one person can fill the role of CEO in a company.

Prerequisite roles for a certain role $R$ are a set of roles which any user should already have before she can be assigned to $R$.

Ahn and Sandhu developed a language to express constraints in RBAC AS00. The language is RCL2000 (Role-based Constraints Language 2000), which is a generalisation of RSL99 (Role-based Separation of duty Language 1999 AS99).

Comment: The safety problem discussed in HRU76 is also interesting to us. Essentially, it is the problem of whether a privilege can be obtained by a subject through finite steps of operations. They have proved that there is no single algorithm that can decide safety problems of this kind for all protection systems. This is so because the situations they considered are extremely general ones in which objects and subjects can be created and destroyed. However, in their paper, they also agree that the problem can be decided when considering restricted cases. In our approach, when a property is verified against a model of an access control system, only models with fixed sizes are considered. Therefore their conclusion of undecidability is not applicable to our approach.
The RBAC model is the most hotly discussed model and is attracting much attention. However, in the RBAC model, privileges cannot be assigned to individuals directly. Rather, they must be associated with roles. In practice, we found that this notion is unrealistic in situations where different individuals have different privileges. For this reason we did not use the RBAC model as our modelling formalism. We created our RW formalism which associates privileges with individuals. The RW formalism and its description language is discussed in Chapter 9 and, there, we shall see how the RW language expresses mechanisms of administration, delegation and constraints.

2.2.4 Languages used for specifying access control policy

Using languages to express access control policies explicitly is another approach to replace access control matrices, as we mentioned in 2.2.3.1. Natural languages are not suitable for specifying access control policies because they are ambiguous in many circumstances and thus too difficult to be analysed as well as interpreted. For this reason, various formal languages were created. These languages usually are precise, terse, and easy to be interpreted and reasoned about. Throughout the rest of this sub-section, we will see several examples of them.

2.2.4.1 XACML

The eXtensible Access Control Markup Language (XACML) is used in our work to translate access control policies written in the RW language. It is proposed by the OASIS (Organisation for the Advancement of Structured Information Standards) committee, intending to be used as a standard language in the field of e-business and other web applications. Another access control policy language published by IBM, called EPLA (PSO), is very similar to it. The XACML used in our work follows the specification described in GM03.

The data-flow model of XACML is described in Figure 4. It is a simplified version of the one in GM03.

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
Figure 2.7: The data-flow model of XACML.

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, such as from the policy information point (PIP), from the environment, from the subjects and the 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 by the PEP and forward it to the PEP.

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 specifies a XML name space declaration (XACML is XML-based), a name for the rule, and the effect of the rule, either Permit or Deny. 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
2.2.4 Languages used for specifying access control policy

1. `<xml version="1.0" encoding="UTF-8">`
2. `<Rule`
3. `xmlns="..."`
4. `RuleId="urn:oasis:names:tc:xacml:rule1"`
5. `Effect="Permit">`

6. `<Description>`
7. A person may read any medical record in the
9. for which he or she is a designated patient
10. `</Description>`
11. `</Rule>`

Figure 2.8: An example for the head and the description of a rule in XACML.

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.

See Figure 2.8 for an example of the head and the description.

An example of the target is shown in Figure 2.9, where the target defines that the rule is
applicable to any request for accessing the namespace specified in line 7. It does not specify
subjects and actions, which means, in this case, any subject is allowed to perform any action.
The designator in line 9 tells the PDP, when the rule is being evaluated, to select the value
designated by the `target-namespace` field in the resource section of the request.

For an example of the condition, see Figure 2.10.

The condition specifies the number of the patient whose records the requester wants to
access must equal to his own, that is, a patient can only access his own records. The function
`string-one-and-only` is used in line 2 and 5 because the designators in line 3 and line 6 may each
return a bag of values. But the `string-equal` function in line 1 only requires two candidates. So
the function `string-one-and-only` guarantees, in each case, the result returned by the designator
contains no more than one value.

Now, suppose a request is received by the PDP, the PDP will first evaluate the applicability
of the rule by checking its target. Following the instruction in line 9 in Figure 2.9, the PDP
will search for the `target-namespace` field in the resource section of the request; select the value
designated by that field; and compare it with the value specified in line 7. If they are equal, the
2.2.4 Languages used for specifying access control policy

```xml
<Target>
  <Subjects><AnySubject/></Subjects>
  <Resources>
    <Resource>
      <ResourceMatch MatchId="...:function:string-equal">
        <AttributeValue DataType="...#string">
          http://www.med.com/record.xsd
        </AttributeValue>
      </ResourceMatch>
    </Resource>
  </Resources>
  <Actions><AnyAction/></Actions>
</Target>
```

Figure 2.9: An example for the target in XACML.

```xml
<Condition FunctionId="...:function:string-equal">
  <Apply FunctionId="...:function:string-one-and-only">
    <SubjectAttributeDesignator AttributeId="...:resource:patient-number">
      <AttributeValue DataType="...#string"/>
    </SubjectAttributeDesignator>
  </Apply>
</Condition>
```

Figure 2.10: An example for the condition in XACML.
condition in Figure 2.10 is further considered. Otherwise, the rule is reckoned as not applicable. To evaluate the condition, following the instructions in line 3 and line 6 in Figure 2.10, the PDP will select the values designated by the patient-number and target-patient-number fields in the resource section of the request, and compare them. If they are equal, the effect of this rule, which is Permit, is returned.

See Figure 2.11 for an example of the requests which are likely to be granted. This request is the kind of request notifications sent by the context handler on link 4 in Figure 2.7. It is also written in XACML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Request ...>
  <Subject><AnySubject/></Subject>
  <Resource>
    <Attribute AttributeId="...:resource:target-namespace"
      DataType="...:#string">
      <AttributeValue>http://www.med.com/record.xsd</AttributeValue>
    </Attribute>
    <Attribute AttributeId="...:resource:patient-number"
      DataType="...:#string">
      <AttributeValue>0001</AttributeValue>
    </Attribute>
    <Attribute AttributeId="...:resource:target-patient-number"
      DataType="...:#string">
      <AttributeValue>0001</AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="...:action:action-id"
      DataType="...:#string">
      <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
</Request>
```

Figure 2.11: A request in XACML.

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 a policy, a rule-combining algorithm must be specified to resolve conflicting results returned
2.2.4 Languages used for specifying access control policy

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*.

2.2.4.2 RDL

The *role definition language* (RDL) [BLM01, HBM98] is proposed with the OASIS (Open Architecture for Security Interworking Services) model [BMY01, BMY03, DDL+98] by the researchers in the Opera group of the Cambridge Computer Laboratory. The language is a formal logic based on Horn clauses. It is used to define role activation conditions, privileges associated with roles, and appointments for OASIS services. Because it is based on Horn clauses, each statement in the RDL may be considered as an axiom in a proof system, aiding both human and automated reasoning. The reason we include the RDL in our discussion is because that we borrow the idea of *parameterised roles* from the RDL and use it in the *RW* language. We will discuss the RDL by first discussing the OASIS model, and then the language itself.

The OASIS model

The OASIS model is a RBAC architecture supported by the Cambridge Event Architecture (CEA [BMB+05, MB98]). It has been developed for achieving secure interoperation of services in open, distributed environments. The motivation of the OASIS project derives from a study on providing ubiquitous access to Electronic Health Records (EHRs) [The97, The98, The00], which are proposed to be used in the National Health Service (NHS) in the UK [BLM01]. Because of the complexities in controlling access to EHRs in distributed environments, the traditional RBAC models [SCFY96] cannot be applied to online medical systems without modifications.

In the OASIS architecture, each OASIS service locates at different places yet connected through networks. Each service is responsible for managing roles used in the service. The responsibilities include creating roles, naming roles, associating roles with privileges and assigning users to roles. Roles may be parameterised, for example, by users’ identities, as required by applications. There is no globally centralised administration of the management of roles. To activate a role in a service, a principal must show credentials to the service to demonstrate that all the activation conditions defined by the local service for activating the role have been fulfilled. During the time of the principle’s occupying the role, if a condition becomes false,
the service will be notified by the CEA, and thus the role will be deactivated subsequently. Appointment is used instead of delegation. Each time a principle wants to appoint a task to another principle, it sends the appointee appointment certificates. The appointee can then use the certificates as credentials to activate the roles necessary to the accomplishment of the task.

Figure 2.12 shows the typical procedure of an OASIS authorisation.

A principal sends credentials to the role entry sub-service, which is responsible for defining roles and role activation conditions based on local policies. A role membership certificate (RMC) is issued to the principal if the credentials satisfy the conditions. A RMC is an encryption-protected capability parameterised by the role name, the identity of the principal and a reference to the service which issues it. The credential records sub-service keeps a record of the RMCs that have been issued by this service. The principal then can present the RMCs to the access control sub-service for validity checking. If the checking is passed, the principal is allowed to use the services provided by the OASIS service.

An example of the interworking between OASIS services is demonstrated in Figure 2.13, where a principal having already obtained RMCs from services A, B and C, activates a new role in service C. The RMCs already obtained serve as prerequisite conditions for obtaining the new RMC. To keep the new role active, all of them must remain valid. Otherwise, if one of them becomes false, service C will be notified by the event channels and thus deactivate the additional RMC issued to the principal.
2.2.4 Languages used for specifying access control policy

![Diagram showing interworking between OASIS services.]

Figure 2.13: The interworking between OASIS services.

The main differences between the OASIS model and the traditional RBAC models [SCFY96] are summarised in the following [BMY03]:

- Roles are service-specific. There is no globally centralised administration of role naming and privilege management.
- Roles may be parameterised.
- Roles remain active if all the prerequisite conditions remain true.
- Appointment replaces delegation.
- The CEA is used to support the validity checking when a role is to be activated. Roles may be deactivated as soon as a particular condition becomes false.
2.2.4 Languages used for specifying access control policy

The RDL

The RDL is used in OASIS to define role activation conditions and privileges associated with roles. We shall see a few examples picked up from [BLM01] and [HBM98].

To express role entry:

\[
\text{SeniorHaemato}logist(x) \leftarrow \text{Haemato}logist(x) \& \text{SeniorDoctor}(x)
\]

The statement states that any principal, denoted by \( x \), intending to activate the role \textit{SeniorHaematologist} must already obtain the roles \textit{Haematologist} and \textit{SeniorDoctor}. Thus, obtaining the roles \textit{Haematologist} and \textit{SeniorDoctor}, is the prerequisite condition for entering the role \textit{SeniorHaematologist}.

To express privileges associated with roles:

\[
\text{append(}\text{haematology field}, y, x) \leftarrow \text{SeniorHaematologist}(x)
\]

The statement states that being a senior haematologist, \( x \) can append data to the field of haematology in a record of patient \( y \). However, it is better to specify that being a doctor, \( x \) can append data to a record of patient \( y \) only if \( y \) is a patient of \( x \). Side conditions can be used for this purpose. See the following statement:

\[
\text{append(}\text{haematology field}, y, x) \leftarrow \text{SeniorHaematologist}(x)\text{[doctor of}(y, x)\text{]}]
\]

To express an appointment:

\[
\text{Examiner}(e) \leftarrow \text{LoggedOn}(p, s)
\]

\[
\text{< ChiefExaminer: } p \text{ in Staff}
\]

The statement states that the \textit{chief examiner} can elect any \( p \) to be an examiner, provided that \( p \) is a staff and \( p \) has logged on host \( s \).

To express an appointment with revocation:

\[
\text{Candidate}(p, e) \leftarrow \text{LoggedOn}(p, s)^*
\]

\[
\text{<" Examiner}(e)^* : (p \text{ in Student})^*
\]

The statement states that the \textit{examiner} \( e \) can appoint a student \( p \) to be a \textit{candidate} in an examination provided \( p \) has logged on host \( s \). The stars denote that if any one of the annotated conditions becomes false, \( p \)'s role of \textit{candidate} will be revoked by the system.
In the design of the RW language, we borrow the idea of parameterised roles from the RDL. For example, in the first statement, the role SeniorDoctor is parameterised by $x$. To have parameterised roles is necessary in situations where the traditional RBAC models are not subtle enough to handle. Consider this following example:

Suppose there is a student information system which maintains students’ marks. In this system, we want to enforce a rule specifying that each student can only read his own marks. To express this rule, only associating the privilege of reading to the role student is not enough. We have to parameterise the role student by each student’s identity. Thus, to express this rule in the RDL, we have:

$$\text{read}(m,x) \leftarrow \text{student}(x)[\text{mark.of}(m,x)]$$

where $m$ denotes a mark and $x$ denotes a student.

### 2.2.4.3 Others

Throughout the rest of this sub-section, several other language-based approaches will be mentioned without thorough discussions. They are relevant to our work, however, their influence on our work is not as much as that of the XACML and the RDL.

**A calculus proposed in [ABLP93]**

The calculus is based on logic. It is used for access control in distributed systems. Roles, groups, resources, ACLs and delegation can be expressed in the calculus. For example, if $A$ and $B$ are two principals, $R$ is a role and $s$ is a statement,

- $A \land B$ states principal $A$ and principal $B$
- $A$ as $R$ states principal $A$ in role $R$
- $B$ for $A$ stands for the principal $B$ obtained when $B$ speaks on behalf of $A$ with appropriate delegation certificates
- $B$ for $A$ says $s$ stands for $B$ says that $A$ says $s$ with proof of delegation

Access control decisions are made based on the result of the evaluation of ACLs and the requesting statements. The problem of undecidability [HRU76] is mentioned, however, the authors argue that access control problems in context are not arbitrary, and thus decidable.
Figure 2.14: Authorisation in trust-management systems.

**D1LP**

Li [Li00] has developed a logic-based language, called *Delegation Logic* (DL), to represent policies, credentials and requests in distributed authorisations. D1LP, the monotonic version of DL, is described in [LGF03].

Access control in large-scale, open, distributed systems is different from that in centralised systems. Authorisations in distributed environments cannot base on the results of authentications because, normally, a requester is unknown to an authoriser before it makes a request. For this reason, the authors of [LGF03] adopt the trust-management (TM) approach [BFL96 BFS96], in which authorisation is viewed as a *proof-of-compliance* problem: Does a set of credentials prove that a request complies with a policy? Because an authoriser does not know a requester before it sends a request, the authoriser often trusts third parties who know the requester better and issue it credentials. The authoriser, therefore, would delegate the authority to determine the identity of the requester to the trusted third parties. Thus, when the requester makes a request, the authoriser will check the credentials carried with the request, and make a decision based on the local policy. This procedure is illustrated in Figure 2.14.

Full details about the syntax and the semantics of D1LP are presented in [LGF03], however, we shall only use an example from online shopping to demonstrate the usage of D1LP.

Suppose, a merchant ShopA will approve a customer’s order if it can determine that the customer has a good credit rating. ShopA trusts BankB and all the parties that BankB trusts in determining credit ratings. BankB believes that a principal has good rating if a credit card company CardC certifies that the principal has an account in good standing.
The policy of ShopA can be expressed in D1LP by the following statement:

ShopA says approveOrder(?U) if ShopA says creditRating(?U, good)

where ?U denotes a variable representing a customer and approveOrder and creditRating are predicates.

The delegation between ShopA and BankB can be expressed by the following statement:

ShopA delegates creditRating(?U, ?R) to BankB

where ?R denotes a variable representing a rating.

The delegation between BankB and CardC can be expressed by the following statement:

BankB says creditRating(?U, good) if CardC says accountGood(?U)

where accountGood is a predicate.

Now, a customer Carl sends an order to ShopA and provides the following credential:

CardX says accountGood(Carl)

ShopA then will generate a query:

ShopA says approveOrder(Carl)?

The query will be approved based on the above policy.

**Cassandra**

Cassandra [BS04a] is a role-based trust management system with an elegant and readable policy specification language based on Datalog with constraints. The language is expressive and adjustable. The expressiveness of the language can be adjusted by choosing an appropriate constraint domain, depending on the requirements of the application. The design process of the language was guided by a case study [BS04b] on proposing an access control policy for a national electronic health record system. In the case study, the policy proposed by the authors contains a total of 310 rules and 58 parameterised roles. The case study demonstrates the applicability of the language to large-scale network environments. Cassandra has also a formal semantics for query evaluation and for the access control enforcement engine. An algorithm for evaluating queries is developed. A prototype of Cassandra is implemented.
2.2.4 Languages used for specifying access control policy

**Ponder**

Ponder [DDLS01] is a declarative, object-oriented language for specifying and managing security policies. The language, as the authors of [DDLS01] state, provides a common means of specifying security policies which can be mapped onto various access control mechanisms in implementation, such as firewalls, operating systems and databases.

A Ponder toolkit is presented in [Dam02]. The toolkit includes a domain browser, a compiler, a policy editor and a management console.

**Tower**

Tower [HV01] is a language for specifying role-based access control policies. The language supports basic structures of RBAC, such as roles, users and permissions, as well as user-role and permission-role relations. Constraints, such as separation of duty, and delegations can also be expressed by the language.

**A language in [SW02]**

Sirer and Wang [SW02] present an access control language for formally specifying and enforcing security policies on web service implementations. The language is based on temporal logic. Security policies written in this language are processed by an enforcement engine, WebGuard, to yield site and platform-specific access control code. This code can then be integrated with a web server and platform-specific libraries to enforce the specified policy on a given web service.

**ASL**

The *authorisation specification language* (ASL), proposed in [JSS97], is a logical language for the specification of authorisations on which a model supporting different access control policies can be based. The language allows users to specify the policy according to which access control decisions are to be made. Policies are expressed by rules which enforce authorisations, conflict resolution, and integrity constraint checking.

**Comment**

XACML is the language that we use to translate policies written in the RW language. We choose XACML because it is standardised, mature and implementable. Sun Microsystems has implemented a software and programming-interface package for XACML [Sun03]. The package contains a simple PDP program which is able to read XACML policies, accept requests...
and make decisions. This package is useful for implementing a real access control system based on existing XACML policies.

The influence that the RDL has on our work is its idea of parameterised roles, which resolves the limitation of the traditional RBAC model. The inclusion of other language-based approaches, which have little influence on our work, in the review is to show that using languages to specify access control policies is often the first step toward the formal analysis of them.

2.3 Model-based verification

Formal verification techniques normally comprise three parts [HR04a]:

- a framework for modelling systems, typically a description language of some sort;
- a specification language for specifying properties of the system to be verified;
- a verification method to evaluate whether the description of the system satisfies the specification.

In terms of how a system is represented, approaches to verification can be classified as proof-based and model-based.

A proof-based approach represents a system using a set of formulas \( \Gamma \) and the specification is another formula \( \phi \). The verification method consists of trying to find a proof that \( \Gamma \vdash \phi \).

A model-based approach represents a system by a model \( M \) for an appropriate logic. The specification is represented by a formula \( \phi \). The verification method consists of computing whether \( M \models \phi \), that is, \( M \) satisfies \( \phi \).

In terms of how a model is used, model-based approaches can be further classified as model-checking and model-finding.

2.3.1 Model-checking

Model-checking starts with the model described by the user, and discovers whether the property is valid on the model. Computations in model-checking are often automatically done by tools, which are known as model-checkers. Model-checking is intended to be used for concurrent\(^1\) reactive\(^2\) systems. It has been proved to be a productive approach in finding bugs in real

\(^1\)A concurrent system is a system in which several processes run simultaneously.
\(^2\)A reactive system is a system which reacts to its environment and is not meant to terminate.
2.3.1 Model-checking

protocols used on the Internet \[\text{Low90}\] and on hardware systems \[\text{BSiRT01}\, \text{MS91}\]. Recently, researchers have started to apply it to the verification of software systems \[\text{HJMS03}\].

Model-checking is based on \textit{temporal logic}. The idea of temporal logic is that a formula is not statically true or false in a model. Instead, a model of temporal logic contains several states and a formula can be true in some state and false in others. Thus, a formula may change its truth value as the system evolves from state to state. Such systems are called \textit{transition systems} in the sense that the state of the system transits from one to another.

A model-checking problem in a transition system can be characterised as follows: Given a model of the system, \(M\); a state of \(M\), \(s\); a formula of temporal logic, \(\phi\); whether \(M, s \models \phi\), that is, \(\phi\) succeeds on the state \(s\) of \(M\).

The common steps of verifying such a problem consists of \[\text{HR04a}\]:

- model the system under consideration using the description language of a model-checker, arriving at a model \(M\);
- code the property using the specification language of the model-checker, resulting in a temporal logic formula \(\phi\);
- run the model-checker with inputs \(M\) and \(\phi\). (\(s\) is usually considered as all initial states.)

The model-checker outputs the answer ‘\text{yes}’ if \(M, s \models \phi\) and ‘\text{no}’ otherwise. If the answer is ‘\text{no}’, most model-checkers also produce a trace of system behaviour which causes this failure.

Temporal logics can be classified according to their view of \textit{time}. \textit{Linear-time} logics think of time as a set of paths, where a path is a sequence of time instances. The \textit{linear-time temporal logic} (LTL) \[\text{Eme91}\] belongs to this class. It assumes implicit universal quantification over all paths. \textit{Branching-time} logics represent time as a tree, rooted at the present moment and branching out into the future. The \textit{computation tree logic} (CTL) \[\text{HR04a}\] is of this kind, which allows explicit existential and universal quantification over all paths. Moreover, there is a third kind: \textit{alternating-time temporal logic} (ATL) \[\text{AHK99}\], which offers selective quantification over those paths that are possible outcomes of games, such as the game in which the system and the environment alternate moves.

In what follows, we will briefly review CTL and its model-checker, SMV; ATL and its model-checker, Mocha, because we once had seriously considered using them but later decided that both of them are not suitable for our approach.
2.3.1 Model-checking

\textbf{MODULE Main}
\textbf{VAR}
\textbf{request : boolean;}
\textbf{status : \{ready, busy\};}
\textbf{ASSIGN}
\textbf{init(status) := ready;}
\textbf{next(status) := case}
\textbf{request : busy;}
\textbf{1 : \{ready, busy\};}
\textbf{esac;}
\textbf{SPEC}
\textbf{AG(request -> AF status = busy)}

Figure 2.15: A SMV program and a specification.

2.3.1.1 CTL and SMV

CTL's model of time is a tree-like structure which brings out the non-deterministic nature of the future. There are different paths leading to the future, any one of which might be the actual path that is realised.

In CTL, we have universal quantifier A which means ‘along All paths’, and existential quantifier E which means ‘along at least (there Exists) one path’. They are used with temporal operators: X, F, G and U, meaning ‘next state’, ‘some Future state’, ‘all future states (Globally)’ and ‘Until’ respectively. For example, AF means ‘some future state along all paths’.

Other conventional logical connectors, such as conjunction and disjunction, are also used in CTL formulas.

The model checker for CTL is called ‘New Symbolic Model Verifier’ (NuSMV) \cite{CCO05}, or sometimes it is simply called ‘Symbolic Model Verifier’ (SMV). It is different from the ‘Cadence SMV’ \cite{McM99a,McM99b}, which is for LTL only.

SMV takes as input a text consisting of a program describing a model and some specifications. It produces as output either the word ‘true’ if the specification holds for all initial states, or a trace showing why the specification is false on the model.

A SMV program is composed of one module called ‘main’ or other modules along with the main module. Variables and their assignments are defined in modules. An assignment usually sets the initial value for a variable and updates its next state's value non-deterministically. For an example of a SMV program and a specification see Figure \ref{fig:smv_example}.
The program in Figure 2.15 defines two variables, request and status. The value of the variable request is not specified. The value of the variable status is defined in two phrases. First, the initial value is set as ready. Secondly, the value of the next state is defined in terms of the current value of the variable request. If request is true, the next value of status is busy. Otherwise, it can be either ready or busy, thus, non-determined. The specification states that ‘for all the states in all paths, if there is a state, say s1, in which request is true then eventually there is a future state along all paths, starting from s1, in which status becomes busy’.

We had considered using the input language of SMV to express access control policies, CTL 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 the SMV input language. Whether an agent can 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 that update the boolean variables’ values. The kind of properties we want to check 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. We extract the strategies by searching for paths among agents’ knowledge states which start from the agents’ initial knowledge state and end up in the set of goal knowledge states. SMV’s ability of temporal reasoning also seems to be suitable for this task. However, despite all these similarities, we cannot use SMV. The reason is illustrated in Figure 2.16.

In Figure 2.16 each point stands for a set of the agents’ knowledge states. Each arrowed line denotes a reading or overwriting step which leads the agents’ knowledge state in one set
to a state in another. Suppose at point A, the agents sample a variable \( q \). The outcome is either true or false and each possible outcome produces a possible route. A strategy is counted as a strategy only if both the routes end up in the set of goal knowledge states, that is, in Figure 2.16 route 1 and route 2 must both succeed on the set of goal knowledge states. A strategy must guarantee the achievability of the goal and therefore, it should not be counted as a strategy if only one route ends up in the set of goal knowledge states. The problem with CTL is that we cannot express that both possible routes generated by a sampling succeed on the set of goal knowledge states.

### 2.3.1.2 ATL and Mocha

The *alternating-time temporal logic* (ATL) is discussed in [AHK99]. While branching-time logics are natural specification languages for closed systems, alternating-time logics are natural specification languages for open systems. A closed system is a system whose behaviour is completely determined by the state of the system. An open system, however, is a system which interacts with its environment and whose behaviour depends on the state of the system as well as the behaviour of the environment.

We can imagine an open system as a game between two players A and B. In each step of the game, each player chooses a move, and the combination of the choices determines a transition from the current state to a successor state. In such a system, we may want to ask a question like this: Is there a strategy for A alone to enforce the system into a certain state no matter how B reacts? Such a question can be expressed by an ATL formula which parameterises paths with agents’ names. For example, in ATL, we can say \( \langle A \rangle \phi \), meaning A is the agent who enforces the system into a state in which \( \phi \) is true.

The model-checker for ATL is called ‘Mocha’ [AdH+]. Like SMV, Mocha takes a program which describes the model of a system as input. The modelling formalism and input language of Mocha is called ‘REACTIVEMODULES’. To perform model-checking, one needs to save specifications written in the form of ATL formulas in a file, and then needs to load the file into Mocha console.

A REACTIVEMODULES program is composed of one or more than one module. Each module defines its variables and atoms. Each variable declared in a module is of one of the following categories: *external* variables, *interface* variables and *private* variables. External variables are the variables declared in other modules. Interface variables are the variables declared within the module and controlled by one atom inside the module. Private variables
2.3.1 Model-checking

module randomwalk
    interface \( x : \text{int} \)

    atom incrdecr
        controls \( x \)
        reads \( x \)
        init
            [true] \( x' := 0 \)
        update
            [true] \( x' := x + 1 \)
            [true] \( x' := x - 1 \)
        endatom
    endmodule

Figure 2.17: A REACTIVEMODULES program.

are the variables whose values are constant. An atom in a module declares variables it controls, reads and awaits. For each controlled variable, it may define evolution rules for upgrading its value. Figure 2.17 shows a REACTIVEMODULES program, which defines a module called ‘randomwalk’. The module defines an interface variable \( x \) and an atom \( \text{incrdecr} \). The atom controls \( x \) and defines its initial value and the updating rules.

We had considered using ATL and Mocha because of Mocha’s agent-based reasoning ability. We can define an access control system in a module and each atom in the module for an agent in the system. However, we finally gave up the idea because of two reasons. From our experience with Mocha [Zha02b], we doubt its ability to model-check big systems. When the system grows large, checkings take long time. Moreover, each variable defined in a module can only be controlled by exactly one atom. This means that if we model an agent-based system, following our modelling scheme, in Mocha, the value of each variable in the system can only be modified by one agent. In practice, we found that this is not convenient.

2.3.1.3 Symbolic model-checking

A major hurdle for model-checking large systems is the state explosion problem. At its most basic level, a model-checking algorithm explores the entire state-space of a system. Suppose there are \( n \) boolean variables in the system, the total number of states of the system, \( N \), is \( 2^n \). As \( n \) grows, even if \( n \) is not a large number, \( N \) will soon become too huge to handle.
2.3.1 Model-checking

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$f(x, y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: The truth table for $\neg(x \lor y)$

Symbolic model-checking \cite{Mcm93} is the most well-known approach for combating this problem, and is the approach we use in our algorithm. This approach represents sets of states symbolically as binary decision diagrams (BDDs). BDDs are directed acyclic graphs which can be used to represent boolean functions. They were introduced by Lee \cite{Lee59} and Akers \cite{Ake78}. Following the work of Bryant \cite{Bry86}, BDDs became popular, for Bryant refined the data structure and developed a set of efficient algorithms for manipulating them.

Using BDDs to represent boolean functions

Boolean functions can be represented by several forms, such as by truth tables, by propositional formulas and by formulas in disjunctive normal form (DNF) or conjunctive normal form (CNF). However none of these representations is comparable to BDDs in terms of compactness, ease of manipulation and straightforwardness of validity and satisfiability checking.

For example, a boolean function $f(x, y)$, represented by the propositional formula $\neg(x \lor y)$, can also be represented by the truth table in Table 2.1.

The truth table is space-inefficient. As the number of variables in the function grows, the total number of lines in a truth table grows rapidly. The propositional formula is space-efficient. However, it is not easy to perform boolean operations on them especially as they grow large and it is not easy to see how they can be satisfied.

To represent the same function in a BDD, one can first represent it in a binary decision tree, which is a simpler form of BDDs (see Figure 2.18(a)).

In the tree shown in Figure 2.18(a), all the non-terminal nodes are labelled with boolean variables $x, y$ and all the terminal nodes are labelled with either 0 or 1. Each non-terminal node has two edges, one dashed line and one solid line. Starting from the root node, we traverse down the tree. Whenever the value of the variable at the current non-terminal node is 1, we

\footnote{This example is taken from \cite{HR04}.}
2.3.1 Model-checking

![Binary Decision Tree Diagram](image)

Figure 2.18: Evolution from a binary decision tree to a binary decision diagram.

![Simple Transition System](image)

Figure 2.19: A simple transition system.

follow the solid line; otherwise, we follow the dashed line. The value of the function represented by the tree is the value of the terminal node we reach.

This tree, however, is not compact enough. There are still duplicate terminals and redundant edges. After removing the duplicate terminals (the result is shown in Figure 2.18(b)) and redundant edges (the result is shown in Figure 2.18(c)), we obtain the BDD representation of the function, which is shown in Figure 2.18(c).

Using BDDs to represent sets of states

Since symbolic model-checking represents sets of states as BDDs, given a finite set of states, $S$, the task is to represent various subsets of $S$ as BDDs. Because BDDs represent boolean functions, what we need is to use boolean functions to denote elements of $S$. Let’s see an example of how to do this. This example is picked from [HR04].

See Figure 2.19 for a simple transition system, where there are two boolean variables $x_1$, $x_2$ and three states $s_0$, $s_1$ and $s_2$. In the state $s_0$, $x_1$ is true and $x_2$ is false. In the state $s_1$, $x_2$ is true and $x_1$ is false. In the state $s_2$, both $x_1$ and $x_2$ are false.

Using propositional formulas to represent boolean functions, we denote the state $s_0$ by
2.3.1 Model-checking

<table>
<thead>
<tr>
<th>Set of states</th>
<th>Representation by propositional formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$x_1 \land x_2$</td>
</tr>
<tr>
<td>${s_0}$</td>
<td>$x_1 \land \neg x_2$</td>
</tr>
<tr>
<td>${s_1}$</td>
<td>$\neg x_1 \land x_2$</td>
</tr>
<tr>
<td>${s_2}$</td>
<td>$\neg x_1 \land \neg x_2$</td>
</tr>
<tr>
<td>${s_0, s_1}$</td>
<td>$(x_1 \land \neg x_2) \lor (\neg x_1 \land x_2)$</td>
</tr>
<tr>
<td>${s_0, s_2}$</td>
<td>$(x_1 \land \neg x_2) \lor (\neg x_1 \land \neg x_2)$</td>
</tr>
<tr>
<td>${s_1, s_2}$</td>
<td>$(\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_2)$</td>
</tr>
<tr>
<td>${s_0, s_1, s_2}$</td>
<td>$(x_1 \land \neg x_2) \lor (\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_2)$</td>
</tr>
</tbody>
</table>

Table 2.2: Representing subsets of states by propositional formulas.

$x_1 \land \neg x_2$, $s_1$ by $\neg x_1 \land x_2$, $s_2$ by $\neg x_1 \land \neg x_2$. Therefore, subsets of states can be represented by the propositional formulas in Table 2.2. Because the formula $x_1 \land x_2$ is not used in other cases, we use it to represent the empty set.

**Using BDDs to represent transition relations**

In order to use BDDs to represent transition relations, we need to use boolean functions to represent transition relations. Consider the transition system of Figure 2.19. Let $\rightarrow$ denote the transition relation and $S$ denote the set of all states. $S$ is $\{s_0, s_1, s_2\}$ and $\rightarrow$ is $\{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}$. We have already seen how subsets of $S$ are represented by boolean functions. Since $\rightarrow$ is a subset of $S \times S$, we need two copies of boolean variables. One set of unprimed variables $x_1, x_2$ denote the current state and one set of primed variables $x'_1, x'_2$ denote the state after the transition. Thus, $\rightarrow$ can be represented by the propositional formula below:

$$(\neg x_1 \land \neg x_2 \land \neg x'_1 \land \neg x'_2) \lor (\neg x_1 \land \neg x_2 \land x'_1 \land \neg x'_2) \lor (x_1 \land \neg x_2 \land \neg x'_1 \land x'_2) \lor (\neg x_1 \land x_2 \land \neg x'_1 \land \neg x'_2)$$

The BDD representation of this formula can be found at Figure 2.20.

**Using BDDs to compute $\text{pre}_3$ and $\text{pre}_4$**

We have seen how BDDs are used to represent subsets of states and transition relations. Suppose $\rightarrow$ is a transition relation and $B_\rightarrow$ is its BDD representation. $X$ is a subset of the set of all states $S$ and $B_x$ is its BDD representation. Now it remains to see how BDDs are used to compute other subsets from $X$ in terms of the transition relation $\rightarrow$. 

41
There are two approaches to obtain other subsets from $X$. We can either compute the sets of states which transition into $X$ or the set of states which states in $X$ transition into. The former is known as the pre-computation and the latter is post-computation. Because what we use in our algorithm is the pre-computation, we will only discuss the pre-computation in what follows.

Given the subset $X$, $\text{pre}_3(X)$ is defined as

$$\{s \in S \mid \text{exists } s' \in S, (s \rightarrow s' \text{ and } s' \in X)\}$$

and $\text{pre}_q(X)$ is defined as

$$\{s \in S \mid \text{for all } s' \in S, (s \rightarrow s' \text{ implies } s' \in X)\}$$

$\text{pre}_q(X)$ can be expressed in terms of complementation and $\text{pre}_3(X)$, as follows: $\text{pre}_q(X) = S - \text{pre}_3(S - X)$, where $S - Y$ denotes the set of all $s \in S$ which are not in $Y$. Therefore, we only need to see how BDDs are used to compute $\text{pre}_3(X)$ in terms of $B_s$ and $B_\rightarrow$. To compute $\text{pre}_3(X)$, one can follow the three steps:

1. Rename the variables in $B_s$ to their primed versions. We use $B_{s'}$ to denote the resulting BDD.
2. Perform logical conjunction on $B_\rightarrow$ and $B_{s'}$. We use $B_{\rightarrow \land s'}$ to denote the resulting BDD.
3. Perform existential quantification on all the primed variables in $B_{\rightarrow \land s'}$. 

42
2.3.1 Model-checking

BDDs packages
BuDDy [CWLN04] and CUDD [Som05] are well-known C implementations for BDDs. However, the one we use is a native Java implementation called ‘JavaBDD’ [Wha04].

2.3.1.4 Abstraction

Abstraction is another well-known technique used along with symbolic model-checking to combat the state explosion problem. It amounts to removing or simplifying details of the original model that are irrelevant to the property under consideration. Verifying the simplified model is evidently more efficient than verifying the original model, however, at the price of potentially incurring wrong results because of the information loss. Abstraction techniques can be distinguished according to how they deal with the information loss. Over-approximation [CGJ94, Kur95] and under-approximation [LPJ+96, PH98] techniques keep the error one-sided.

Over-approximation techniques release constraints, and thus enriching the behaviour of the system. As a result, the correctness in the abstract model implies the correctness in the original one. In contrast, under-approximation techniques remove irrelevant behaviour of the system so that a specification violation at the abstract level implies a violation of the original system. Abstraction is typically a manual process.

Clarke, Grumberg, Jha, Lu and Veith [CGJ+03] introduced a fully automatic abstraction technique, known as the CounterExample-Guided Abstraction Refinement (CEGAR). This method starts with a relatively small skeletal representation of the system to be verified, and computes increasingly precise abstract representations of the system according to the information extracted from spurious counterexamples due to over-approximation.

Comment  Our work conforms to the classic approach of model-checking in that the RW formalism serves as the underlying modelling formalism, the RW language as the modelling language and the RW model-checking algorithm implemented by AcPeg as the verification method.

The RW algorithm adopts a symbolic approach where sets of states and transition relations are represented as BDDs and pre-computations are done through BDDs. Therefore, the discussion in 23.1.3 serves as a background for the discussion of the algorithm in Chapter 5.

CEGAR is used by AcPeg, along with a three-level over-approximation abstraction technique (see 5.3 for more details).
2.3.2 Model-finding

Model-finding differs from model-checking in that it finds models in accordance with the user’s description but forming counterexamples to the user’s specifications. The Alloy analyser is a classical example of the implementations in this approach.

2.3.2.1 Alloy

The Alloy modelling language is discussed in [Jac02a] and [JSS01]. The latest version of the Alloy analyser is 3.0 beta [Jac04]. However, one can still find instructions on how to use the stable 2.0 version in [Jac02b] and [JSS00].

Alloy is a declarative, structural modelling language, based on first-order logic. The main differences between the model-finding approach adopted by the Alloy analyser and typical model-checking approaches used by most model-checkers are summarised in Table 2.3.

For an example of a description and a specification defined in Alloy see Figure 2.21. The syntax is based on the 3.0 version.

```alloy
module filesystem
    sig FSObject { parent : lone Dir}
    sig Dir extends FSObject {
        contents: set FSObject
    }
    fact defineContents {
        all d: Dir, o: d.contents | o.parent = d
    }
    sig File extends FSObject {}
    fact fileDirPartition {
        File + Dir = FSObject
    }
    assert acyclic {
        no d: Dir | d in d."contents
    }
    check acyclic for 5
```

Figure 2.21: An Alloy model and a specification.

The description in Figure 2.21 describes a simple file system. The signature FSObject is the top-level signature in this example. It defines a relation parent which maps every object in FSObject to zero or one object in Dir. This relation is inherited by all sub-signatures extending
### 2.3.2 Model-finding

<table>
<thead>
<tr>
<th>Motivation of design</th>
<th>Alloy</th>
<th>Model-checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designed for addressing the complexity that arises from relational structures.</td>
<td>Designed for addressing the complexity that arises from concurrency.</td>
<td></td>
</tr>
</tbody>
</table>

| Compositional ability | An Alloy model only has one state machine. It has no or little compositional ability. | A model in model-checking may contain several state machines running in parallel. It is compositional. |

| Declarative or operational | Models in Alloy are declarative in that they describe a system's state by explicitly listing properties and constraints of the system. | Like programming languages, models in model-checking are operational in that they explain how states are constructed and how execution obtains a new state from an old one. |

| Ability to express constraints | Constraints are easily stated in Alloy models. | Modelling formalisms in model-checking normally do not provide mechanisms to state constraints explicitly. |

| Ability to build complex data structures | Alloy can build complex data structures, such as links. | Modelling formalisms in model-checking normally only provide abilities to build low-level primitive data structures. |

| Ability to do temporal reasoning | Alloy does not do temporal reasoning. | Model-checking does temporal reasoning. |

| Ability to reason about partial structures | Alloy can be used to analyse partial structures in which only part of the system's behaviour is defined. | Model-checking cannot be used to analyse partial structures. |

Table 2.3: Differences between Alloy and model-checking.

FSObject. The signature Dir extends FSObject. The relation contents relates every object in Dir to a set of objects in FSObject. The File signature also extends FSObject. The fact fileDirPartition constrains the behaviour of the system in that the sum of objects in File and Dir must be equal to the number of objects in FSObject. The assertion acyclic introduces a specification which states that no object in Dir can be found in its content paths. The check statement checks the validity of the assertion in models which contain no more than 5 objects in the top-level signature FSObject.

We had also considered using Alloy in our work. However, two reasons made us finally dismiss the idea. Alloy nearly has no ability to do temporal reasoning. If one wants to do
reachability checking in Alloy one has to hard-code states and transition relations. But this makes the checking very inefficient and slow. Also, from our experience with Alloy we find that Alloy, when checking large systems, becomes very slow. Nevertheless, we borrow the ideas of populating defined classes and checking within fixed sizes from Alloy.

2.4 Related work on reasoning about access control

Although using formal methods to analyse access control policies is becoming popular, tool-based approaches are scarce. Therefore, in what follows, we broaden the scope by including works relating to both formal analysis and access control, however not necessarily tool-based, for discussing. Related work on either access control or model-based verification has been discussed in section 2.2 and 2.3 respectively.

2.4.1 Model-checking access control policies

The formalism called RW [GRS04] which is our starting point provides abilities to express fine-grained conditional authorisation, which means that authorisation is based on arbitrary conditions rather than solely on roles. This enables the RW formalism to model a wide range of access control systems. Another novelty about the RW formalism is its built-in ability to expresses permissions about permissions and delegation. As we have discussed in 2.2.3 permissions about permissions and delegation are important solutions to decentralise the administrative power and thus to avoid the root-bottleneck problem.

The mathematical description of the RW language is proposed in the paper. The language can be used to describe access control policies modelled in the RW formalism and goals of agents. These goals are typically reading and writing values of resources. The authors also propose a decision procedure which determines whether a given coalition of agents has the means to achieve a given goal.

The work presented in this thesis can be regarded as a continuation and development of the work described in [GRS04]. Our work uses the RW formalism as the underlying modelling formalism. We adopt the description part of the RW language proposed in [GRS04] and create the specification part, because the syntax for expressing goals in the original RW language is too limited. We make the language machine-readable by having written a compiler for it with the help of JavaCC [Java03]. Our model-checking algorithm uses some ideas from the decision procedure proposed in [GRS04], however, as it now stands, the algorithm is already
very different from that procedure. Given a coalition of agents and a goal, that procedure can only decide whether the goal is achievable. The algorithm presented in this thesis can also figure out how the goal can be achieved.

### 2.4.2 Analysing access control policies through MTBDDs

In [FKMT03], Fisler, Krishnamurthi, Meyerovich and Tschantz present Margrave, a software suite for analysing role-based access control policies. Margrave reads policies in XACML, translating them into multi-terminal decision diagrams (MTBDDs [CFM+02]). Properties can then be verified with the help of these MTBDDs. 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. Moreover, XACML policies lend themselves to various forms of boolean representations which makes it natural to represent them using MTBDDs. For example, suppose there is a policy, saying: “In a department of a university, faculty can assign grades and students can receive grades.” The MTBDD representation for this policy is shown in Figure 2.22.

Margrave verifies whether a policy preserves a property by taking a query which expresses the property as input and outputting an answer to the query. It traverses the MTBDD representing the policy, using the information provided in the query and seeing which terminal it gets to.
2.4.3 Reasoning about access control policies using first-order logic

Change-impact analysis is also an important aspect of their work. Margrave can take two policies 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 representing policies take from 70ms to 335ms. Memory consumption is about 4.7Mbytes. Because MTBDDs scale up quite well, their tool is quite capable to handle large cases.

2.4.3 Reasoning about access control policies using first-order logic

Halpern and Weissman [HW03] demonstrate how a fragment of first-order logic can be used to represent and reason about access control policies. Starting from this fragment, they end up with a language which is quite expressive yet tractable.

A rule saying ‘all librarians can edit the catalogue’ can be characterised by the following statement:

$$\forall x(\text{Librarian}(x) \rightarrow \text{Permitted}(x,\text{edit catalogue}))$$

A rule saying ‘any one who is allowed to sing may dance’ can be characterised by the following statement:

$$\forall x(\text{Permitted}(x,\text{sing}) \rightarrow \text{Permitted}(x,\text{dance}))$$

A prototype which has a user graphic interface and a reasoning engine is implemented. The user graphic interface allows non-logicians to enter rules, such as ‘each student can read book No.3.’ Through theorem proving the engine answers whether a request should be granted based on the policy statements.

2.4.4 Others

Schaad and Moffett [SM02] use Alloy to specify a RBAC-style model, ARBAC97-style extensions and a set of separation of duty properties. They analyse possible conflicts that could arise out of decentralised administrative actions with respect to the separation of duty constraints. Through their work, they demonstrate the general suitability of the Alloy language for specification and analysis of role based systems.

Guttman, Herzog, Ramsdell and Skorupka [GHR04] present a systematic way to analyse access control policies in the Security-Enhanced Linux system (SELinux). They develop a highly abstract model for the access control mechanism adopted by the SELinux operating
system. In this model, the system configuration determines a transition system which represents possible information flows. Properties of the system take the form of information flow security goal statements which describe the objectives that SELinux is intended to achieve. The goal statements are written in LTL. They have tools which take the transition system and produce input for NuSMV. 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 [GH04].

Raimondi and Lomuscio [RL04] investigate the problem of the verification of epistemic properties of multi-agent systems via model-checking. Epistemic properties of multi-agent systems express a variety of attitudes of agents, such as their knowledge, beliefs, desires, as well as their temporal evolution. In the paper, they focus on agents’ knowledge. They provide a model-checking algorithm, which uses BDDs, for checking those epistemic properties. They also present a software package which implements the algorithm. They demonstrate the usability of the software by means of a traditional example from the security literature – the dining cryptographers [Cha88].

Chess [Che02] presents Eau Claire, a tool which is used to find security flaws in C programs. Using the method of static checking, Eau Claire reads programs and specifications that define security properties. Eau Claire translates the program’s source code into a series of verification conditions and presents them to an automatic theorem prover. It then uses this theorem prover to look for certain types of errors.

Comment The work of Fisler, Krishnamurthi, Meyerovich and Tschantz [FKMT05] is so far the most comparable work to ours. Their tool reads XACML code, transforms the code into MTBDDs and performs analysis. However, in their paper, they didn’t discuss the topic of uncovering weaknesses in access control policies caused by interactions of rules, co-operations between agents and multi-step actions.
Chapter 3

The $RW$ formalism and its description language

3.1 Outline

The $RW$ formalism is formally defined in 3.2. The steps of using the $RW$ formalism to model an access control policy are demonstrated in 3.3 through an example. The modelling power of the $RW$ formalism is discussed in 3.4. The syntax and the semantics of the description part of the $RW$ language are explained in 3.5.

3.2 Definition

**Definition 3.1** Let $L(P)$ 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 $(\Sigma, P, \tau, \omega)$, where $\Sigma$ is a set of agents, $P$ is a set of propositional variables and the mappings $\tau, \omega : P \times 2(\Sigma) \to L(P)$ 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 $\tau(p, \{a\})$ and $\omega(p, \{a\})$, respectively. We assume that access rights are exercised by one agent at a time in this thesis for the sake of simplicity and write $\tau(p,a)$ instead of $\tau(p,\{a\})$, and similarly for $\omega$. Thus the formulas $\tau(p,a),\omega(p,a) \in L(P)$ define the conditions under which $a$ is permitted to read and overwrite the value of $p$. They are functions on $S$’s states.
3.3 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 \( a, b \in \text{Agent} \), \( p \in \text{Paper} \), \( P \) includes:

- \( \text{author}(p, a) \) is an author of \( p \)
- \( \text{pcmember}(a) \) is a PC member
- \( \text{chair}(a) \) is the chair of the PC
- \( \text{reviewer}(p, a) \) is assigned to PC member \( a \) for reviewing
- \( \text{subreviewer}(p, a, b) \) is assigned by PC member \( a \) to sub-reviewer \( b \)
- \( \text{submittedreview}(p, a) \) a review for \( p \) has been submitted by (sub-)reviewer \( a \)
- \( \text{review}(p, a) \) the reviewing result for \( p \) from \( a \), either accepted or rejected

For each \( p \in P \), \( a \in \text{Agent} \), \( r(p, a) \) and \( w(p, a) \) are defined using variables in \( P \) and logical connectors: \( \neg \) (negation), \( \land \) (conjunction), \( \lor \) (disjunction), \( \rightarrow \) (implication), \( \exists \) (existent 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 true. \( \Sigma \) denotes the set of agents. ‘\( \equiv \)’ denotes ‘is defined as’):

\[
\begin{align*}
\text{r}(\text{author}(p, a), x) &\equiv \top, \text{r}(\text{pcmember}(a), x) \equiv \top, \text{r}(\text{chair}(a), x) \equiv \top & \text{rule 1} \\
\text{w}(\text{pcmember}(a), x) &\equiv \text{chair}(x) \lor (\text{pcmember}(x) \land a = x) & \text{rule 2} \\
\text{r}(\text{reviewer}(p, a), x) &\equiv \text{pcmember}(x) \land \neg \text{author}(p, x) & \text{rule 3} \\
\text{w}(\text{reviewer}(p, a), x) &\equiv (\text{chair}(x) \land \text{pcmember}(a) \land \neg \text{author}(p, a) \\
&\lor (\exists d \in \Sigma \text{ subreviewer}(p, x, b)) & \text{rules 4-5} \\
\text{r}(\text{subreviewer}(p, a, b), x) &\equiv (\text{pcmember}(x) \land \neg \text{author}(p, x)) \lor x = b \lor x = a & \text{rule 6} \\
\text{w}(\text{subreviewer}(p, a, b), x) &\equiv (\text{reviewer}(p, x) \land \neg \text{author}(p, b) \\
&\land x = a \\
&\lor (\exists d \in \Sigma (\text{ subreviewer}(p, a, d) \\
&\lor \text{subreviewer}(p, d, b) \lor x = b)) \\
&\land \neg \text{submittedreview}(p, x)) & \text{rules 7-8} \\
\text{r}(\text{submittedreview}(p, a), x) &\equiv \text{pcmember}(x) \land \neg \text{author}(p, x) & \text{rule 9}
\end{align*}
\]

52
3.4 Discussions on the \( RW \) formalism

\[
\begin{align*}
\text{w(submittedreview}(p,a),x) & := \left( \left( (x = a) \land \left( \exists b \in \Sigma \text{ subreviewer}(p,b,x) \right) \right) \land \neg \text{submittedreview}(p,x) \right) & \text{rule } 10 \\
\text{r(review}(p,a),x) & := \left( \left( \text{pcmember}(x) \land \neg \text{author}(p,x) \right) \land \text{submittedreview}(p,a) \right) \land \text{reviewer}(p,x) \land \left( \left( \exists b \in \Sigma \text{ subreviewer}(p,b,x) \right) \land \left( \text{reviewer}(p,x) \lor \text{submittedreview}(p,x) \right) \right) & \text{rule } 11 \\
\text{w(review}(p,a),x) & := \left( (x = a) \land \neg \text{submittedreview}(p,x) \right) \land \left( \exists b \in \Sigma \text{ subreviewer}(p,b,x) \lor \text{reviewer}(p,x) \right) & \text{rule } 12
\end{align*}
\]

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

3.4 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 [AVPS03], 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’ [SBM96] or ‘meta-policies’ [BM02]. 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 5.1 \( w(\text{reviewer}(p,a),x) \) defines the permission for changing \( \text{reviewer}(p,a) \), 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 \[\text{Bar02}\]. Delegation is a solution which leads to the avoidance of root bottleneck. The \textit{RW} formalism’s ability to express delegation is demonstrated by the way that a reviewer can appoint sub-reviewers in Example \ref{example:sub-reviewers}.

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 \[\text{SCFY96}\]. Constraints can be expressed by the \textit{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.

\section{The description language}

The \textit{RW} language is a machine-readable language designed for expressing access control policies modelled in the \textit{RW} formalism and properties to be verified. A complete model written in the \textit{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 a policy. The run-statement populates the defined classes, and thus creates a concrete instancing model of the defined policy. The specification specifies a property to be verified. In this section, we discuss issues on the syntax and the semantics of the program, that is, the description part which expresses a policy. We leave the discussion on the run-statement and the specification to Chapter \[\text{III}\].

\subsection{Example 3.1 written in the \textit{RW} language}

Before formally discussing the syntax and the semantics of the description part, we shall see, in Figure \ref{fig:policy-rw}, the \textit{RW} script for the policy defined in Example \ref{example:sub-reviewers}.

\subsection{Syntax and semantics of the description part}

A complete definition of the syntax can be found in Appendix \[\text{A}\]. There 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.
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) | (pcmembre(a) and a=user));
}
reviewer(p, a){
   read : pcmember(user) & ~author(p, user);
   write : (chair(user) & pcmembre(a) & ~author(p,a))
      or ((pcmembre(user) & user=a & reviewer(p,user))
      & ~E b: Agent [subreviewer(p,user,b)]);
}
subreviewer(p, a, b){
   read : (pcmembre(user) & ~author(p,user)) or user=b or user=a;
   write : (reviewer(p,a) & ~author(p,b) & user=a
      & ~E d: Agent [subreviewer(p,a,d) or 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

Figure 3.1: The RW script for the policy defined in Example 3.1
3.5.2 Syntax and semantics of the description part

All the grammatical units are enclosed by ( and ). Here, following the script in Figure 3.4, 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.

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 Figure 3.4 has only one class defined, which stands for the set of papers.

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

```plaintext
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 \( p \in \text{Paper} \text{ and } a \in \text{Agent} \), \( \text{author}(p, a) \) 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
instance 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 Figure 3.1)

```c
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 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 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.
Chapter 4

Properties and their \textit{RW} specifications

4.1 Outline

The syntax and the semantics of the run-statement is described in \[12\]. Issues on how to specify a property for verifying in the \textit{RW} language is discussed in \[13\] including several examples.

4.2 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:

\begin{verbatim}
(RunStatement) ::= “run for” (NumberClassPair) (“,” (NumberClassPair))*
(NumberClassPair) ::= ⟨Integer⟩ ⟨ClassName⟩
\end{verbatim}

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 \cite{Jac02}. 

For the system defined by the program in Figure 6.1 if the following line

\begin{verbatim}
run for 3 Paper, 4 Agent
\end{verbatim}
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 author(paper : Paper, agent : Agent) is instantiated to twelve propositional variables: from author(p_1, a_1) to author(p_3, a_4). After the population, the total number of the propositional variables in \( P \) is 104.

4.3 Properties

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

\[
\text{check } \{ \text{L} \mid \varphi \rightarrow A : \psi \}
\]

where \( \text{L} \) defines a number of quantified variables used in \( \varphi, \psi \) and \( A; \varphi \) (optional) is a list of conditions based on which the goal, defined by \( \psi \), 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 \( \psi \) on the basis of the conditions defined by \( \varphi \).

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 the 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 \( \text{L} \varphi, \psi, A \) separately, and finally we shall see a few example queries.

4.3.1 Quantified variable definition

Quantified variables in \( \text{L} \) 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 \texttt{dij} is used.

4.3.2 Conditions

Conditions defined in \( \varphi \) 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 \( \varphi \) are connected
by logical conjunction (\( \land \)). For a \( p \in P \) which occurs in \( \varphi \), we summarise its possible forms
and their meanings in the following list. (\( \top \) stands for the boolean value \texttt{true} and \( \bot \) stands
for \texttt{false}.)

\[
\begin{align*}
\begin{array}{ll}
p^* & p's \ value \ is \ constant \ during \ a \ checking, \ however, \ this \ value \ is \ unknown \ by \\
& the \ agents \ in \ A \\
\neg p & p \ is \ \top(\bot) \ at \ the \ beginning \ of \ a \ checking \ and \ there \ is \ at \ least \ an \ agent \ in \ A \\
& who \ knows \ p \ is \ \top(\bot) \ at \ the \ beginning \ of \ the \ checking \\
\neg p^* & p \ is \ \top(\bot) \ constantly \ during \ the \ course \ of \ a \ checking \ and \ there \ is \ at \ least \ an \\
& agent \ in \ A \ who \ knows \ p \ is \ \top(\bot) \ at \ the \ beginning \ of \ the \ checking \\
\end{array}
\end{align*}
\]

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 4.3.2) 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 \( \psi \) can be a \textit{simple goal} or a \textit{nested goal}. A simple goal is a combination
consisting of conjunction and disjunction of three kinds of atomic goals. These are \textit{making}
goals, \textit{realising} goals and \textit{reading} goals, written using '{ }', '{ }' and '{ [ ]}'; respectively. If \( l_1, l_2 \)
and \( l_3 \) are propositional formulas belonging to the set \( L(P) \) defined in Definition 4.1, \( \{ l_1 \} \) is the
\textit{goal of making} \( l_1 \) \textit{true}; \( \langle l_1 \rangle \) is the \textit{goal of realising} that \( l_1 \) \textit{is true}; and \( [l_1] \) is the \textit{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 \( \text{not} \ l_1 \) and \( \{ l_3 \} \), which means: achieve either \( l_2 \) or \( l_3 \) according to whether
4.3.4 Coalition of agents

A single 'realising' goal \( l_1 \) 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

\[
\text{check} \{ L \parallel \varphi \rightarrow A_1 : (\psi_1 \text{ AND } A_2 : (\psi_2 \ldots \text{ AND } A_n : (\psi_n)\ldots))\}
\]

where \( \psi_1, \psi_2, \ldots, \psi_n \) are simple goals. Its meaning is: Are there strategies or guessing strategies available for agents in \( A_1, A_2, \ldots, \text{ and } A_n \), such that, if conditions in \( \varphi \) are true, the agents in \( A_1 \) can achieve the goal \( \psi_1 \) and then the agents in \( A_2 \) can achieve the goal \( \psi_2, \ldots \), and finally the agents in \( A_n \) can achieve the goal \( \psi_n \). What this nested goal describes is a sequencing of actions performed by the agents in \( A_1, A_2, \ldots, \text{ and } A_n \). However, this sequencing is achieved only if the conditions defined in \( \varphi \) do not enable agents in \( A_i \) to make other subgoals true while on the way to achieving \( \psi_i \).

4.3.4 Coalition of agents

In the case of a simple goal, \( A \) defines the coalition who intends to achieve the goal \( \psi \). In the case of a nested goal, each \( A_i \) defines the coalition who is to achieve \( \psi_i \).

4.3.5 Example queries written in the \textit{RW} language

In what follows we shall see a few queries (together with run-statements) written in the \textit{RW} language for the system defined by the program in Figure 3.11

**Query 4.1**

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

The query asks: Are there strategies or guessing strategies available for \( a \in \text{Agent} \) 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**

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

The query asks: Are there strategies or guessing strategies available for \( c \in \text{Agent} \) 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 \( p \).
4.3.5 Example queries written in the RW language

Query 4.3
run for 1 Paper, 3 Agent
check {E disj a,b,c: Agent | chair(c)! & ~submittedreview(p,a)! & submittedreview(p,b)! & ~author(p,a)! & pmember(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 [4.1]. 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 [4.1] defined in Example [4.3] 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 ~pmember(a)!

-> {c}:{[pmember(a)] AND {a}:{[~pmember(a)] AND {c}:{[pmember(a)]}}

AND {a}:{[~pmember(a)] AND {c}:{[pmember(a)]}})

This is a five-level nested query which asks: Are there strategies or guessing strategies available for \(a,c \in \text{Agent}\) 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.
Chapter 5

RW model-checking

5.1 Outline

The RW model-checking problem is defined in 5.2. The assumptions made by the algorithm are listed and justified in 5.3. The transition system built upon the agents' knowledge states is explained in 5.4. The boolean representation for \( k_{\text{init}} \) and \( K_e \) are discussed in 5.5 and 5.6 respectively. The backwards reachability computation is described in 5.7. The pseudo-code for the algorithm is presented in 5.8. Its correctness is proved in 5.9 and its complexity is discussed in 5.10.

5.2 The RW model-checking problem

Given a query \( Q \), as the one defined in 1.3, 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.

5.3 Assumptions made by the algorithm

Given an access control system \( S(\Sigma, P, r, w) \), a query \( Q \), which includes the conditions \( \phi \), the goal \( \psi \) 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 carrying out a strategy are:

63
1. While carrying out the strategy, the agents are assumed to completely know the policy of $S$.

2. At the beginning of executing the strategy, $A$ holds no knowledge about the initial state of $S$, except for the values of the variables marked by '*' in $\varphi$.

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 whether $a$ 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 $w(p, a)$ is true and $a$ knows $w(p, a)$ 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 $\psi$, therefore, in each of its steps, 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 $S$.

### 5.4 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 making goals involved 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 $K_0$. Each step of a strategy takes $A$ from a knowledge state to a possibly richer one until a state in $K_0$ is reached. A knowledge state combines knowledge of the initial state of the...
5.4 The transition system

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 just been 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 $k_{\text{init}}$ to one in $K_0$ 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

$$v_p \quad \text{is true if A knows the initial value of } p$$
$$t_p \quad \text{is true if A knows initially } p \text{ is true}$$
$$v_p \quad \text{is true if A knows the current value of } p$$
$$t_p \quad \text{is true if A knows currently } p \text{ is true}$$

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

$$V_0 = \{ p \in P \mid v_0p \text{ is } \top \}, \quad T_0 = \{ p \in P \mid t_0p \text{ is } \top \}$$
$$V = \{ p \in P \mid v_p \text{ is } \top \}, \quad T = \{ p \in P \mid t_p \text{ is } \top \}$$

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 Figure 5.1.

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 regard its goal as having been reached. Figure 5.2 illustrates the above process, however in a simplified situation where only one variable $p$ is considered.
5.5 Representation of $k_{\text{init}}$

Note the transition relations 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 $k_{\text{init}}$ to $K_a$ the algorithm works backwards by inverting the process described above. Before discussing the backwards reachability computation, we shall first see how $k_{\text{init}}$ and $K_a$ are represented by 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.

5.5 Representation of $k_{\text{init}}$

According to our assumptions in Section 5.3, $k_{\text{init}}$ 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 $\varphi$. This means that for all variables in $P$ which also occur in $\varphi$ 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 $v_0, t_0, v_p, t_p$ to describe $A$'s initial and current knowledge on it. What we need to do is to use a boolean expression composed of the knowledge variables to represent $k_{1414}$. 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 $\varphi$ and are marked by $!$ and by $*$. $P^-$ is the set for variables in $P$ which only occur negatively in $\varphi$ and are marked by $!$ and by $*$. $P^o$ is the set for all the other variables in $P$. Now for each $p \in P^+$, we use $(v_0, t_0, v_p, t_p)$ to represent $A$'s initial knowledge on it. For each $p \in P^-$, we use $(v_0, t_0, v_p, t_p)$ to represent $A$'s initial knowledge on it. For each $p \in P^o$, we use $(v_0, t_0, v_p, t_p)$ to represent $A$'s initial knowledge on it. Therefore, the representation of $k_{1414}$ by the knowledge variables is the conjunction of all the representations of the above forms for variables in $P$.

### 5.6 Representation of $K_G$

Given a formula $\psi$ describing the goal we want to produce a representation of the goal states. $\psi$ is a conjunctive and disjunctive combination of reading goals $[l]$, realising goals $<l>$, and making goals $\{l\}$, where $l$ in each case is a boolean combination of variables of $P$.

We want to represent $\psi$ 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 $\{v_0, t_0, v_p, t_p \mid p \in P\}$. To relate this to the notation of Figure 5.1, note that $v_0$ is true in a state if $p \in V_0$, $t_0$ 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 $\{v_0, t_0, v_p, t_p \mid p \in P\}$ expressing the agent's 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 $\bot$ is substituted for $p$ and another in which $\top$ 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

\[
(l[t_p/p] \land v_p) \lor (l[\top/p] \land l[\bot/p] \land \neg v_p)
\]
Since we use this kind of formula frequently, we define a notation for it.

**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

$$\gamma_{V,T} l = (l[t_p/p] \land v_p) \lor (l[\top/p] \land l[\bot/p] \land \neg v_p)$$

expresses the agent’s ability to determine that $l$ is true.

### 5.6.1 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_0, t_0, v_p, t_p | p \in P\} is $\gamma_{V_0,T_0} l \lor \gamma_{V_0,T_0}(\neg l)$.

### 5.6.2 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_0, t_0, v_p, t_p | p \in P\} is $\gamma_{V_0,T_0} l$.

### 5.6.3 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_0, t_0, v_p, t_p | p \in P\} is $\gamma_{V,T} l$.

### 5.7 Backwards reachability computation

#### 5.7.1 Computing sets of states

To find strategies the algorithm starts from $K_0$, searching for sets of the knowledge states which transition into $K_0$ by overwriting any $p \in P$ to true; sets of the knowledge states which transition into $K_0$ by overwriting any $p \in P$ to false; and sets of the knowledge states which transition into $K_0$ 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 $k_{init}$ is found in a set of knowledge states, the goal is considered as reachable by following the operations represented
5.7.1 Computing sets of states

by the transition relations which connect the set in which \( k_{\text{init}} \) is found to \( K_0 \). 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 pre-sets. For any \( a \in A \), \( p \in P \) and a given set of knowledge states \( Y \),

\[ \text{Pre}_{p\rightarrow}^{a} (Y) \]

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:

\[ \{(V_0,T_0,V,T) \mid \exists (V'_0,T'_0,V',T') \in Y, \ V'_0 = V_0, T'_0 = T_0, V' = V \cup \{p\}, T' = T \cup \{p\}, \gamma_V;\tau(p,a) = \top\} \]

\[ \text{Pre}_{p\leftarrow}^{a} (Y) \]

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:

\[ \{(V_0,T_0,V,T) \mid \exists (V'_0,T'_0,V',T') \in Y, \ V'_0 = V_0, T'_0 = T_0, V' = V \cup \{p\}, T' = T \setminus \{p\}, \gamma_V;\tau(p,a) = \top\} \]

\[ \text{Pre}_{p\rightarrow}^{-a} (Y) \]

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 finding out it is true (\( \top \)). Its formal definition is:

\[ \{(V_0,T_0,V,T) \mid \exists (V'_0,T'_0,V',T') \in Y, \ p \notin V_0, p \notin T_0, p \notin V, p \notin T, V'_0 = V_0 \cup \{p\}, T'_0 = T_0 \cup \{p\}, V' = V \cup \{p\}, T' = T \cup \{p\}, \gamma_V;\tau(p,a) = \top\} \]

\[ \text{Pre}_{p\leftarrow}^{-a} (Y) \]

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 finding out it is false (\( \bot \)). Its formal definition is:

\[ \{(V_0,T_0,V,T) \mid \exists (V'_0,T'_0,V',T') \in Y, \ p \notin V_0, p \notin T_0, p \notin V, p \notin T, V'_0 = V_0 \cup \{p\}, T'_0 = T_0 \setminus \{p\}, V' = V \cup \{p\}, T' = T \setminus \{p\}, \gamma_V;\tau(p,a) = \top\} \]

In the above definitions, \( \gamma_V;\tau(p,a) = \top \) denotes the condition under which \( a \) knows with Knowledge \( V \), \( T \) that she is permitted to read/overwrite \( p \). The mapping \( \tau(p,a) \) 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 \( \tau(p,a) \), we need to use the knowledge variables in \( V \) and \( T \) to replace every occurrence of variables in \( \tau(p,a) \), because variables in \( V \) and \( T \) describe \( A \)’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 as the principle we use in representing \( K_0 \).

In \( 5.6 \) we have seen how \( K_0 \) is represented by a boolean expression. The algorithm uses BDDs to represent boolean expressions. Following the formula given in [HR04], all the pre-sets
can be obtained from \( K_0 \) through BDD computations (see the three steps discussed in \[3.1.3\]) using the formula:

\[
\text{Pre}_{\rightarrow, \alpha}(Y) = \text{exists}(\hat{\gamma}', \text{apply}(\text{and}, B_{\rightarrow}, B_{\rightarrow'}))
\]

where \( B_{\rightarrow'} \) 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 \( \hat{\gamma}' \) stands for all the primed variables. In our case, \( B_{\rightarrow} \) is obtained by synthesising all the conditions in the formal definition of \( \text{Pre}_{\rightarrow, \alpha} \) discussed above.

## 5.7.2 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 \( s \) of the pair \((Y, s)\) denotes the fact that \( s \) is a strategy that enables \( A \) to reach \( K_0 \) from each state in \( Y \). For \( K_0 \), the \( s \) is simply ‘skip’, which means ‘do nothing’.

The algorithm starts with the pair \((K_0, \text{skip})\). The core of the algorithm works as follows: Given the pair \((Y, s)\), it adds the pairs \((\text{Pre}_{\rightarrow, \alpha}(Y), (p := \top; s)) \) and \((\text{Pre}_{\rightarrow, \alpha}(Y), (p := \bot; s))\).

For any two pairs \((Y_1, s_1)\) and \((Y_2, s_2)\), it adds the pair \((\text{Pre}_{\rightarrow, \alpha}(Y_1) \cap \text{Pre}_{\rightarrow, \alpha}(Y_2), \text{if } (p) \text{ by } a \text{ then } s_1 \text{ else } s_2)\).

The algorithm continues until no new pairs are generated. Now, all the pairs whose set of knowledge states contains \( k_{\text{init}} \) 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 \( \gamma_{Y, T}(p, a) = \top \) when computing \( \text{Pre}_{\rightarrow, \alpha}(Y) \) and \( \text{Pre}_{\rightarrow, \alpha}(Y) \).

## 5.8 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 \( k_{\text{init}} \) and the set of goal knowledge states \( K_0 \). It outputs at least a strategy for going from \( k_{\text{init}} \) to some state in \( K_0 \). The algorithm works by backwards reachability from \( K_0 \) to \( k_{\text{init}} \). It maintains a set of states it has seen, called \text{states seen}, and a data structure storing the pairs found so far, called \text{strategies}.

Input: \( K_0 \) - set of goal knowledge states \( k_{\text{init}} \) - 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: at least a strategy for going from \( k_{\text{init}} \) to some state in \( K_0 \) if such strategies exist

1. \( \text{strategies} := \emptyset \);
2. \( \text{states} \_\text{seen} := \emptyset \);
3. put \( (K_0, \text{skip}) \) in \( \text{strategies} \);
4. repeat until \( \text{strategies} \) does not change{
   5. choose \( (Y_1, s_1) \) \( \in \) \( \text{strategies} \);  // for all pairs in \( \text{strategies} \)
   6. for each \( p \in \mathcal{P} \{
      7. for each \( a \in A \{
         8. \quad \text{\( PTY_1 := \text{Pre}^{3,a}_{p \rightarrow \top}(Y_1) \);} 
         9. \quad \quad \text{if } ((PTY_1 \neq \emptyset) \land (PTY_1 \not\subseteq \text{states} \_\text{seen}))\{
            10. \quad \quad \quad \text{\( \text{states} \_\text{seen} := \text{states} \_\text{seen} \cup PTY_1 \);} 
            11. \quad \quad \quad \text{\( p_{s_1} := \text{”set } p \text{ to } \top \text{ by } a;” + s_1 \);} 
            12. \quad \quad \quad \text{\( \text{strategies} := \text{strategies} \cup (PTY_1, p_{s_1}) \);} 
            13. \quad \quad \quad \text{if } (k_{\text{init}} \in PTY_1) 
            14. \quad \quad \quad \text{output } p_{s_1}; 
            15. \quad \quad \} 
         16. \quad \text{\( PFY_1 := \text{Pre}^{3,a}_{p \rightarrow \bot}(Y_1) \);} 
         17. \quad \quad \text{if } ((PFY_1 \neq \emptyset) \land (PFY_1 \not\subseteq \text{states} \_\text{seen}))\{
            18. \quad \quad \quad \text{\( \text{states} \_\text{seen} := \text{states} \_\text{seen} \cup PFY_1 \);} 
            19. \quad \quad \quad \text{\( p_{s_1} := \text{”set } p \text{ to } \bot \text{ by } a;” + s_1 \);} 
            20. \quad \quad \quad \text{\( \text{strategies} := \text{strategies} \cup (PFY_1, p_{s_1}) \);} 
            21. \quad \quad \quad \text{if } (k_{\text{init}} \in PFY_1) 
            22. \quad \quad \quad \text{output } p_{s_1}; 
            23. \quad \} 
      7. \} 
   6. \} 
   5. \} 
26. choose \( (Y_2, s_2) \) \( \in \) \( \text{strategies} \);  // for all pairs in \( \text{strategies} \)
27. for each \( p \in \mathcal{P} \{
   28. \quad \text{for each } a \in A \{
   29. \quad \quad \text{\( PSY := \text{Pre}^{3,a}_{p \rightarrow \top}(Y_1) \land \text{Pre}^{3,a}_{p \rightarrow \bot}(Y_2) \);} 
   30. \quad \quad \text{if } ((PSY \neq \emptyset) \land (PSY \not\subseteq \text{states} \_\text{seen}))\{
   31. \quad \quad \quad \text{\( \text{states} \_\text{seen} := \text{states} \_\text{seen} \cup PSY \);} 
   32. \quad \} 
23. \} 
24. \}
strategies := strategies ∪ (PSY, pss);

pss := “if (p) by a then s1 else s2”;

if (k_{init} ∈ 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 attach 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_{init} will not be found in any set generated during the
precomputations. 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, it is because Algo-0
is easier to reason about than Algo-1 that we present Algo-0 here and use it as the basis for
the following analysis.

5.9 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 nec-
essarily s).

Proof: The algorithm performs exhaustive backwards reachability and therefore will find all
states for which there is a strategy to arrive at \( K_0 \).

Since \( s \) is a strategy from \( k_{init} \) to \( K_0 \), \( k_{init} \) will be in the set of states found and therefore
the algorithm will output a strategy from \( k_{init} \) to \( K_0 \). ⊣
5.9 Proof of correctness

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 \( k_{\text{init}} \) to \( K_0 \) the algorithm finds at least one of them.

Proof: Following Lemma 11 however the choice operator is resolved, \( k_{\text{init}} \) will eventually be included in states_seen, and therefore some strategy will be generated. \( \Box \)

Lemma 2 For all \((Y, s)\) \( \in \) strategies, and for all \( y \in Y \), \( s \) succeeds on \( y \) and the result is in \( K_0 \).

Proof: We look at all the ways that \((Y, s)\) can be added to strategies. At the beginning, \((K_0, \text{skip})\) 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 three circumstances:

(i) \((PTY_1, pt s_1)\) is added, where, \( \exists a \in A \) and \( p \in P \), such that \( PTY_1 = \text{Pre}_{\mu = \top}^{a, 3} (Y_1) \), \( pt s_1 \) = “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 \( K_0 \). 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 \( pt s_1 \) succeeds on all the states in \( PTY_1 \) and the result is in \( K_0 \).

(ii) \((P FY_1, pf s_1)\) is added, where, \( \exists a \in A \) and \( p \in P \), such that \( P FY_1 = \text{Pre}_{\mu = \perp}^{a, 3} (Y_1) \), \( pf s_1 \) = “set \( p \) to \( \perp \) by \( a \)” + \( s_1 \), and \((Y_1, s_1)\) is in strategies.

The argument for the above case applies also to this one.

(iii) \((PS Y, ps s)\) is added, where, \( \exists a \in A \) and \( p \in P \), such that \( PS Y = \text{Pre}_{\mu = \top}^{a, 3} (Y_1) \cap \text{Pre}_{\mu = \perp}^{a, 3} (Y_2) \), \( ps s \) = “if \( (p) \) by a then \( s_1 \) else \( s_2 \)” 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 \( K_0 \), and \( y_2 \in Y_2 \), \( s_2 \) succeeds on \( y_2 \) and result is in \( K_0 \). We also know for all \( y \in PS Y \) that \( a \) can read \( p \) and if it is \( \top \), the result of that is in \( Y_1 \). However, if it is \( \perp \), the result of that is in \( Y_2 \). Therefore \( ps s \) succeeds on all the states in \( PS Y \) and the result is in \( K_0 \).
5.10 Computational complexity

Theorem 3 If the algorithm outputs the strategy \( s \) then \( s \) succeeds on \( k_{\text{init}} \) and the result is in \( K_0 \).

Proof: From Lemma 2 we know that for all \((Y, s) \in \text{strategies} \) and \( y \in Y \), \( s \) succeeds on \( y \) and the result is in \( K_0 \). Because if \( s \) gets output, there must exist a \( Y \), such that \( k_{\text{init}} \in Y \) and \((Y, s) \in \text{strategies} \). Therefore, it follows that \( s \) succeeds on \( k_{\text{init}} \) and the result is in \( K_0 \). \( \triangleright \)

From the implication of theorem 3 we know that if there is no strategy \( s \) which succeeds on \( k_{\text{init}} \) and results in \( K_0 \), the algorithm will output none.

5.10 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 are made in advance.

- For two BDDs \( B_1 \) and \( B_2 \), the complexity of the operation apply(and/or, \( B_1 \), \( B_2 \)) 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 apply(and/or, \( B_X \), \( B_Y \)) in the worst case is \( 2^{8|P|} \).

- Suppose \( B_{\tau} \) is a BDD representing one of the transition relations presented in \( F \); the number of nodes in \( B_{\tau} \) 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 whose operations take 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 \( |K| (2^{|P|}) \) pairs in strategies. Therefore, this step repeats at most \( 2^{|P|} \) times.

**Line 6** This step repeats \(|P|\) times.

**Line 7** This step repeats \(|A|\) times.

**Line 8** According to the formula for computing \( \text{Pre}^3_\omega(Y) \) in 5.7.1, the operation of this step involves the BDD computation \( \text{apply}(\text{and}, B_{\mu-	au}, B_{Y'}) \) followed by an existential quantification over all the \( V', T' \) variables on the resulting BDD of the \( \text{apply} \) operation. Let \( |B_{\mu-	au}| \) denote the number of nodes in \( B_{\mu-	au} \) and \( |B_{Y'}| \) denote the number of nodes in \( B_{Y'} \). \( 2^{|P|} \) is the total number of \( V', T' \) variables. The complexity of the \( \text{apply} \) operation in the worst case is \( 16|P|2^{|P|} \) and the complexity of the existential quantification is \( 2^{|P|} \). The worse one of the two is \( 16|P|2^{|P|} \). Therefore, the time spent on this step in the worst case is determined by \( 16|P|2^{|P|} \).

**Line 9** The time spent on this step is determined by the time spent on checking if \( PTY_1 \) is a subset of \( \text{states} \_\text{seen} \). Checking if \( PTY_1 \) is a subset of \( \text{states} \_\text{seen} \) can be done by forming the union of \( PTY_1 \) and \( \text{states} \_\text{seen} \) and then seeing if the resulting set is equal to \( \text{states} \_\text{seen} \). Hence the time spent on this step in the worst case is determined by \( 2^{|P|} \).

**Line 10** The time spent on this step in the worst case is determined by \( 2^{|P|} \).

**Line 16** The time spent on this step in the worst case is determined by \( 16|P|2^{|P|} \).

**Line 17** The time spent on this step in the worst case is determined by \( 2^{|P|} \).

**Line 18** The time spent on this step in the worst case is determined by \( 2^{|P|} \).

**Line 26** This step repeats at most \( 2^{|P|} \) times.

**Line 27** This step repeats \(|P|\) times.

**Line 28** This step repeats \(|A|\) times.
5.10 Computational complexity

**Line 29** The time spent on this step in the worst case is determined by $16|P|2^{|P|+1} + 2^{|P|}$.

Counting the larger one of the two, the time is determined by $2^{|P|}$.

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

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

Adding the time spent on each step together, we get $2^{|P|} \times (|P| \times |A| \times (16|P|2^{|P|} + 2^{|P|} + 2^{|P|} + 16|P|2^{|P|} + 2^{|P|} + 2^{|P|} + 2^{|P|} \times |P| \times |A| \times (2^{|P|} + 2^{|P|} + 2^{|P|} + 2^{|P|}))$. Therefore, the complexity of the algorithm in the worst case is asymptotically bounded above by $3 \times 2^{|P|}$. However, as the experimental data in Table 5.31 show, the situation in practice is far better than this.
Chapter 6

Implementation

6.1 Outline

The functions of AcPeg and its platform-dependency are described in §6.2. The notion of computational round is clarified in §6.3. The difference between a strategy and a guessing strategy is demonstrated through the checking of an example query in §6.4. The mechanism of abstraction, including CEGAR, is explained in §6.5. Several strategies for nested queries are shown in §6.6. A case study of an employee information system is presented in §6.7. A case study of a student information system is presented in §6.8. A case study of a patient record system is presented in §6.9. The performance of the tool is discussed in §6.10.

6.2 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 Chapter 7). A three-level abstraction mechanism is built into the tool to work with the RW model-checking. CEGAR (CounterExample-Guided Abstraction Refinement [CGJ+08]) 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 [Whe04]. 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.
6.3 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 122 there are three quantified variables: \( a, c \) (disjointed) are defined on the set \( \text{Agent} \), and \( p \) is defined on the set \( \text{Paper} \). The set \( \text{Agent} \) is populated to four elements \( \{a_1, a_2, a_3, a_4\} \) – and the set \( \text{Paper} \) 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 \( \text{disj} \) 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 \( \text{disj} \), 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 3 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.

6.4 Strategies and guessing strategies

The difference between a strategy and a guessing strategy is clarified using the following example.

**Example 6.1** Following Definition 3.1 we define an access control system \( S(\Sigma, P, \tau, \omega) \), where \( P = \{u, x, y, z\} \) and \( \Sigma = \{a\} \). For each \( p \in P \), mappings \( \tau(p, a) \) and \( \omega(p, a) \) are summarised by Table 6.1.
6.4 Strategies and guessing strategies

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>w</td>
<td>T</td>
<td>¬u</td>
<td>u</td>
<td>x ∨ y</td>
</tr>
</tbody>
</table>

Table 6.1: Mappings r(p, a) and w(p, a) for the access control system in Example 6.4

The RW program for this system is shown in Figure 6.1

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

Figure 6.1: The RW script for the policy defined in Example 6.4

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 4) 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 Figure 6.2 where [p=1 a=1] stands for 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.

\[
\begin{align*}
\text{[p=1 a=1]} \\
\text{Acting agents: [1]} \\
\text{Guessing strategy: 1} \\
\text{if (u(1) is true) by 1 { \\
\quad \text{set y(1) to true by 1; } \\
\quad \text{set z(1) to false by 1; } \\
\quad \text{skip; } \\
\text{} } \\
\text{else { \\
\quad \text{set x(1) to true by 1; } \\
\quad \text{set z(1) to false by 1; } \\
\quad \text{skip; } \\
\text{}}}
\end{align*}
\]

The number of guessing strategies found is: 1

Figure 6.2: The guessing strategy found for Example 6.1

6.5 Abstraction

To enable AcPeg to handle large cases, we added mechanisms of abstraction to the tool. During the course of a checking, the most time-consuming computations are using BDDs to represent the conditions such as \( V' = V \cup \{ p_k \} \) in computing the pre-sets. Such conditions represent the fact that when an action, either sampling or overwriting, is performed on \( p_k \), it only changes \( A \)'s knowledge on \( p_k \) – 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, \( V' = V \cup \{ p_k \} \) is represented by \( (\bigwedge_{j=1, j \neq i}^{[P]} ((v_{p_j} \land v_{p_j}) \lor (-v_{p_j} \land -v_{p_j})) \land v_{p_i} \). If the number of \( P \)'s elements 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 on all the variables in \( P \); when an action is performed only on \( p_k \). For instance, if we only track \( A \)'s knowledge on \( p_k \) when an action is performed on \( p_k \), the above expression representing \( V' = V \cup \{ p_k \} \) 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.
6.5 Abstraction

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 \( \psi \). 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 \( \psi \), 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 [CGJ+03].

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 for causing the finding of the strategy or guessing 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 Figure [6.6].

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 \texttt{author(1,1)} is not tracked during the course of the checking. Following the suggestion provided by AcPeg, we put the variable \texttt{author(1,1)} into
[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

Figure 6.3: A false strategy and a suggestion made by AcPeg when checking Query 1.2 in
abstraction level 2 and CEGAR.

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.

6.6 Strategies for nested queries

The strategy described by Strategy 1.1 is found by AcPeg when Query 1.3 is checked. The
strategy is shown in Figure 6.4

The strategy described by Strategy 1.2 can be found if the following query is checked by
AcPeg: (Query 1.2 differs from Query 1.3 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, not only does AcPeg find the strategy described by Strategy 1.2 it 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
Figure 6.5
6.7 Case study – an employee information system

\[
\begin{align*}
[a=1 & \ b=2 & \ c=3 & \ p=1] \\
\text{Strategy: 4,1} \\
\text{Coalition: [1]} \\
\text{if (review(1,2) is true) by 1 { } } \\
\text{skip;} \\
\text{else { } } \\
\text{skip; } \\
\text{Coalition: [1, 3]} \\
\text{set reviewer(1,1) to true by 3; } \\
\text{set submittedreview(1,1) to true by 1; } \\
\text{skip; }
\end{align*}
\]

Figure 6.4: The strategy found for Query \([4,3]\)

The strategy found by AcPeg, when Query \([4,3]\) is checked, is shown in Figure 6.6.

6.7 Case study – an employee information system

Example 6.2 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 day-off, 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 reading 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, \(\Sigma\) be the set of employees and thus \(P\) includes the following propositional variables, for all \(b \in \text{Bonus}\), \(a, a_1, a_2 \in \Sigma\):

- \(\text{bonus}(a, b)\) \quad \text{bonus option } b \text{ is owned by } a
- \(\text{manager}(a)\) \quad a \text{ is a manager in the department}
- \(\text{director}(a)\) \quad a \text{ is the director of the department}
- \(\text{advocate}(a_1, a_2)\) \quad \(a_2\) is \(a_1\)’s advocate
6.7 Case study – an employee information system

[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;

Figure 6.5: Two strategies found for Query 5.2

The permission mappings \( r \) and \( w \) can be defined as follows:

\[
\begin{align*}
    r(\text{bonus}(a,b), x) & \equiv \\
    & \left( x = a \lor \text{director}(x) \right) \\
    & \lor \left( \text{manager}(x) \land \neg \text{manager}(a) \lor \neg \text{director}(a) \lor \text{advocate}(a,x) \right) \\
\end{align*}
\]

rule 1

\[
\begin{align*}
    w(\text{bonus}(a,b), x) & \equiv \\
    & \left( \text{manager}(x) \land \neg \text{manager}(a) \lor \neg \text{director}(a) \lor \text{director}(x) \right) \\
\end{align*}
\]

rule 2

\[
\begin{align*}
    r(\text{manager}(a), x) & \equiv \text{true} \\
\end{align*}
\]

rule 3

\[
\begin{align*}
    w(\text{manager}(a), x) & \equiv \\
    & \left( \text{director}(x) \lor \left( x = a \land \text{manager}(a) \lor \neg \text{director}(a) \lor \neg \text{manager}(a) \lor \neg \text{director}(a) \lor \text{director}(x) \right) \right) \\
\end{align*}
\]

rule 4

\[
\begin{align*}
    r(\text{director}(a), x) & \equiv \text{true} \\
    r(\text{advocate}(a_1,a_2), x) & \equiv \text{true}
\end{align*}
\]

rule 5,6

84
[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;

Figure 6.6: The strategy found for Query 6.3

\( w(\text{advocate}(a_1, a_2), x) \equiv \left( x = a_1 \lor \left( \text{advocate}(a_1, a_2) \land x = a_2 \right) \right) \) rule 7

The RW script for this example is shown in Figure 6.7

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 \( a_1, a_2, a_3 \in \Sigma \) are different agents. \( a_1 \) and \( a_2 \) are managers. \( a_3 \) is the director. \( a_1 \) first resigns her managerness; then \( a_2 \) set \( a_1 \)'s bonus; and after that \( a_3 \) promotes \( a_1 \) to be manager again. This strategy is found when Query 6.3 is checked by AcPeg.

**Query 6.3**

run for 4 BONUS, 8 Agent
checkE disp j 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))})

85
6.8 Case study – a student information system

Example 6.3 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 $\Sigma$ be the set of agents and thus $P$ includes the following propositional variables, for all $a, a_1, a_2 \in \Sigma$:

- $\text{student}(a)$: $a$ is a student
- $\text{lecturer}(a)$: $a$ is the lecturer
- $\text{higher}(a_1, a_2)$: student $a_1$ is in a year higher than that of student $a_2$
- $\text{demonstrator_of}(a_1, a_2)$: student $a_1$ is a demonstrator of student $a_2$
- $\text{mark}(a)$: student $a$’s mark (Here, in a highly abstract way, we think of a student’s mark as either pass or fail)

We assume that the relation $\text{higher}$ is antisymmetric and transitive. Note, however, that currently there is no way in the RW language to specify such constraints. (See §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 an inappropriate interpretation of the $\text{higher}$ relation. The user needs to be aware of this.
6.9 Case study – a patient record system

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

Figure 6.9: The RW script for the access control system in Example 6.3

The RW script for the model is in Figure 6.4.

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 6.4 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 6.4

run for 3 Agent
check(E disj 1,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.

6.9 Case study – a patient record system

The policy defined in this example is based on the case study of the Multidomain Healthcare System in [BMY01].
Example 6.4 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, a1, a2 ∈ Σ:

- 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(a1, a2)  a2 is on the exclusion list of patient a1
- 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(a1, a2)  a1 is a treating doctor of patient a2

The RW script for the model is in Figure 6.10.

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 6.5

Query 6.5
run for 6 Agent
check(E dinj p,d: Agent || treating_doctor(d,p)! and patient(p)*)!
-> {d}::{"treating_doctor(d,p)" AND {d}::{record(p)!)}

Query 6.5 was checked by both Algo-0 and Algo-1 and no strategy was found.

6.10 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 5.10).

In Table 6.10 we summarise the data on the memory usage and the time spent on all the testing cases discussed in the previous sections of this chapter. All the checkings 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 usability of the tool is also 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 [6.3]).

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 Query [6.3] and assign different elements to the set Bonus and the set Agent. The experimental results are summarised
### 6.10 Performance

| Query   | Assignment | $|P|$ | Memory usage | Time spent |
|---------|------------|-----|--------------|------------|
| A       | Paper=3 Agent=4 | 104 | 156 156 | 3600 3600 |
| B       | Paper=3 Agent=3 | 27  | 155 155 | 250 250  |
| C       | Paper=3 Agent=3 | 27  | 155 155 | 1000 700  |
| D       | Paper=3 Agent=8 | 27  | 155 155 | 250 252  |
| E       | Bonus=3 Agent=8 | 112 | 162 162 | 30000 22500 |
| F       | Agent=10     | 230 | 156 156 | 33807 34181 |
| G       | Agent=8      | 160 | 156 156 | 41979 41284 |

Table 6.2: Experimental results obtained from running the test cases in Chapter 6.

| Assignment | $|P|$ | Memory usage | Time spent |
|------------|-----|--------------|------------|
| Bonus=3 Agent=8 | 24  | 155 155 | 324 288  |
| Bonus=3 Agent=10 | 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 |

Table 6.3: Experimental results obtained from checking Query 6C3 with different sizes of the model.

From the data in Table 6C3 we can see that, in practice, the computing time increases far slower than the estimation under the worst case discussed in 5.10. 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 [Jac02b]. 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.
Chapter 7

RW to XACML

7.1 Outline

The translation is explained in [7.2]. Instructions on how to set up a database to support the access-decision making based on the converted XACML policy file are spelt out in [7.3].

7.2 Compiling RW to XACML and SQL

7.2.1 Structure of the converted XACML file

AcPeg 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 is converted to a rule in XACML plus a default rule which denies everything. The structure of the output XACML file is shown in Figure 7.1.

The title of the policy contains information about this policy – its name space, identity and the identity of the rule-combination algorithm used by this policy. 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 returns the result Permit. 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 set 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 each rule, except the last default one, is Permit. Together with the permit-overrides algorithm, the policy denies whatever is not explicitly permitted.
7.2.1 Structure of the converted XACML file

```xml
<?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>
```

Figure 7.1: The structure of an output XACML policy file.
For the conference paper review system defined in Example 4.1, 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 RIV 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 Figure 7.2. The rule contains a Target tag, which evaluates applicable situations for this rule.

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

Figure 7.2: Correspondence between a privilege definition and its generated 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 Figure 7.2). 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 Figure
7.2.1 Structure of the converted XACML file

```xml
<Request>
  <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>
```

Figure 7.3: A request for the rule in Figure 7.2

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 = a1`, 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 `Action` 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 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 the 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 Figure 7.4.

```
reviewer(p, a) {
    read : pcdmember(user) & author(p, user);
}
```

```xml
<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 pcdmember WHERE agent=requester) AND NOT EXISTS (SELECT * FROM author WHERE paper=arg_paper AND agent=requester));</AttributeValue>
  </Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
    <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject: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>
</Rule>
```

Figure 7.4: A formula in a RW script and its generated Condition tag.

AcPeg produces SQL code that looks up the truth values of the variables in P in a database. For instance, given an a ∈ Agent, the truth value of pcdmember(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 its converted SQL statement. The external function self-defined:evaluate-sql, which appears as a conditional function in Condition tags, evaluates SQL statements. The mapping SQL from RW formulas to SQL statements described below defines
the correspondence implemented by AcPeg.

\[ \text{SQL}(\text{pcmember}(a)) = \text{EXISTS (SELECT * FROM pcmember WHERE agent=arg\_agent}) \]

This is an example for the simplest case, in which the condition contains just one predicate
with one parameter. The truth value of the condition is equivalent to the non-emptiness
of the selection result. The column name \textit{agent} is derived from the definition of the
predicate \text{pcmember}(\text{agent} : \text{Agent}). Thus the table \text{pcmember} must contain a column
named \textit{agent}. The string \textit{arg\_agent} is the formal name of parameter \textit{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.

\[ \text{SQL}(R(a_1, \ldots, a_n)) = \text{EXISTS (SELECT * FROM R WHERE R}_{a_1}=a_1' \text{ AND } \ldots \text{ AND } R_{a_n}=a_n') \]

If the predicate has more than one parameter, the SQL selection condition is a conjunction.
Again, the truth value of the \textit{RW} condition is equivalent to the non-emptiness of the
selection result. Here \textit{R}_{a_1} \ldots \textit{R}_{a_n} stand for the column names derived from the definition of
predicate \textit{R}. \textit{a}_1' \ldots \textit{a}_n' are the formal names for \textit{a}_1 \ldots \textit{a}_n.

\[ \text{SQL}(a_1 = a_2) = (a_1'=a_2') \]

The translation of equality formulas in \textit{RW} is straightforward. \textit{a}_1' and \textit{a}_2' are the formal names for \textit{a}_1 and \textit{a}_2.

\[ \text{SQL}(-f) = \text{NOT SQL}(f), \text{SQL}(f_1 \land f_2) = (\text{SQL}(f_1)) \text{ AND (SQL}(f_2)) \]

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.

\[ \text{SQL}(\exists x \in D \, f) = \text{EXISTS (SELECT id FROM D AS D\_id WHERE SQL(f))} \]

Elements are selected from the table \textit{D}, which is supposed to list all the elements from the \textit{RW} class
with the same name, and \textit{f} is evaluated for each of them. The \textit{RW} condition \exists x \in D \, f
holds if and only if the resulting selection is non-empty. The string \textit{id} is the default
name for a column in tables describing defined classes. The alias \textit{D\_id} is given to \textit{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 \textit{RW} formula, the result of SQL is prefixed \textbf{SELECT
* FROM T WHERE}. Here \textit{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 \textit{f} is \textbf{SELECT * FROM T
WHERE (SQL(f))};
7.2.3 The external function – self-defined:evaluate-sql

The above clauses allow any RW formula to be translated into an SQL statement. Figure 7.1 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 Figure 7.1 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 any paper and the value selected by the ResourceAttributeDesignator on agent to replace every occurrence of the string any 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 support the access-decision making. Figure 7.2 below illustrates the idea of how we set up a 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.
7.3 Assumptions about the database

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

<table>
<thead>
<tr>
<th></th>
<th>col</th>
<th>id</th>
<th>id</th>
<th>paper</th>
<th>agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>yes</td>
<td>a1</td>
<td>p1</td>
<td>a1</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a2</td>
<td>p2</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 7.5: Tables and the structure of the database set up for the conference example.

The conditions are given as follows:

- 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.
Chapter 8

Discussions and conclusions

8.1 Summary

In this thesis 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 $\mathcal{L}$, the RW formalism can be used to model policies used by a wide variety of access control systems. As the case studies in Chapter 6 and Chapter 7 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 instance 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 6.3 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 $\text{student}(a)$ being true excludes the possibility of $\text{lecturer}(a)$ being true. Although in Example 6.3 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 6.4 for an agent $a$, the value of $\text{chair}(a)$ being true implies the value of $\text{pmember}(a)$ 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 General steps of using the $RW$ framework to verify security properties

We summarise the general steps of using the $RW$ framework to verify security properties in the following list:

1. Identify an access control problem which consists of a possible security failure and the circumstances in which the failure is likely to happen.

2. Model the access control rules which apply to those circumstances, capturing the aspects that relate to the potential security failure and ignoring those unrelated ones.

3. Write the $RW$ program for the rules and specify the property that expresses the potential security failure in the $RW$ language.

4. Run AcPeg to perform the $RW$ model-checking.

5. Examine the strategies output (if there are some) to identify the problems, if the checking results demonstrate the likely existence of the failure.
8.4 Amending policies

review(p, a){
    read : pmember(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)]);
}

Figure 8.1: The amended version of rule 3.1 in Example 3.1

6. Amend the policy according to the information provided by the strategies output and
check the property again against the amended model.

8.4 Amending policies

Outputting strategies helps 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 Figure 8.1 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 Figure 8.1 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 Figure 8.1 exists lies in rule 3.1 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 Figure 8.1 where the condition
is added at the end.

The reason that the second strategy in Figure 8.1 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 ∈ Agent, p ∈ Paper, we have:

\[ \text{assigned}(p, a) \quad p \text{ was once assigned to PC member } a \text{ for reviewing} \]
8.5 On the translation to XACML

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

Figure 8.2: The amended version of rule 4 in Example 8.1

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

Figure 8.3: The accessing rules for the predicate assigned(paper : Paper, agent : Agent).

We then change rule 4 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 Figure 8.2.

For access rules to the predicate assigned(paper : Paper, agent : Agent), see Figure 8.3. 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 6.1 is constructed. Now, checking Query 6.3 (the condition ~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.

8.5 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. CM77.

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. \textit{RW} permission conditions written using quantifiers can meaningfully apply to systems with varying sets of resources and agents and our implementation can handle this.

8.6 Future work

A deeper study of the situations where the \textit{RW} approach can be applied is likely to be an important topic which needs further investigation. This investigation should include a better understanding of the security regulations adopted in entities in the real world, such as hospitals, companies, banks and universities, and a more comprehensive recognising of the potential security vulnerabilities that exist in their policies. This study will make us understand the needs of the real world more clearly and thus more efficiently adapt our approach to meet the challenges.

Another possible direction of future work is to enrich the expressive power of the \textit{RW} language, so that the limitation discussed in 8.2.2 can be removed. To meet this aim, simply adding mechanisms to the language to express constraints is not enough. Corresponding functions which ensure that the constraints to be respected during the checking must be added to AcPeg as well.

Furthermore, implementing a friendly and easy-to-use graphical user interface for AcPeg may also be a thread of future work.
Appendix A

Syntax of the RW language

(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))
(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)
A Syntax of the RW language

\[
\begin{align*}
\text{(Specification)} & ::= \text{(CheckStatement)} \\
\text{(CheckStatement)} & ::= \text{“check” } ']' ']' \text{ (QuantifiedVariablesList) } ']' \\
\text{(Conditions)} & ::= \text{(Condition) } ']' (\text{Condition}) ']' \\
\text{\textbf{Condition}} & ::= \text{\textbf{PositiveCondition}} | \text{\textbf{NegativeCondition}} \\
\text{\textbf{PositiveCondition}} & ::= \text{\textbf{PredicateName}} ']' ']' (\text{\textbf{FormalParameter}}) ']' (\text{\textbf{Goal}}) ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']' ']
\end{align*}
\]
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 are 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 whether $k_{init}$ is in the set or not. Algo-1 adds a pair constructed from a newly found set to strategies if $k_{init}$ 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 := {};
put (K_0, skip) in strategies;
repeat until strategies does not change{
    choose (Y_1, s_1) ∈ strategies    // for all pairs in strategies
    for each p ∈ P{
        for each a ∈ A{
            PTY_1 := Pre_{p,a}^{}(Y_1);
            if ((PTY_1 ≠ ∅) ∧ (PTY_1 ∉ any set of the pairs in strategies)){
                pts_1 := “set p to ⊤ by a;” + s_1;
                if (k_{init} ∈ PTY_1)
                    output pts_1;
                else
                    strategies := strategies ∪ (PTY_1, pts_1);
            }
        }
    }
}
```
\( PFY_1 := \text{Pre}_{p_1}^{a_1}(Y_1); \)

if ((\( PFY_1 \neq \emptyset \)) \& (\( PFY_1 \not\subseteq \text{any set of the pairs in strategies} \))){

\( pfs_1 := \) "set \( p \) to \( \bot \) by \( a_1" + s_1; \)

if (\( k_{\text{init}} \in PFY_1 \))

output \( pfs_1 \);

else

strategies := strategies \cup (PFY_1, pfs_1);

}

}

choose \((Y_2, s_2) \in \text{strategies} \) \quad // for all pairs in strategies

for each \( p \in P \{

for each \( a \in A \{

PSY := \text{Pre}_{p+}^{a}(Y_1) \cap \text{Pre}_{p}^{a}(Y_2); \)

if ((\( PSY \neq \emptyset \)) \& (\( PSY \not\subseteq \text{any set of the pairs in strategies} \))){

\( pss := \) "if \( p \) by \( a \) then \( s_1 \) else \( s_2"; \)

if (\( k_{\text{init}} \in PSY \))

output \( pss \);

else

strategies := strategies \cup (PSY, pss);

}

}

}
Appendix C

Command line parameters

The format of the command that executes AcPeg is

```
java RWcheck parameters filename [abstraction level]
```

where parameters is a list of characters that specify the behaviour of AcPeg; filename is the name (path may be included) of the RW script that 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 as the script. The translated XACML file has the same name as 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 ‘e’, ‘0’ and ‘1’ are mutual-exclusive in the command line.
Bibliography


BIBLIOGRAPHY


[BS00] Ezedin Barka and Ravi Sandhu. A role-based delegation model and some extensions. In NISSC, 2000. 2.2.3 2.2.3.1


[CCO+05] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, and Marco Roveri. NuSMV 2.2 user manual, 2005. This document and the model checker can be obtained from http://nusmv.irst.it. 2.3.1.1


[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM), 50:752–794, Sep 2003. 1.5 2.3.1.2 5.2 6.5


[com93] OASIS committee, founded in 1993. Information about the OASIS committee can be found at [http://www.oasis-open.org](http://www.oasis-open.org)


[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In ICSE’05, St. Louis, Missouri, USA, May 2005. 2.2.2 [2.2.2 2.2.3 2.2.4]


[GM03] Simon Godik and Tim Moses. eXtensible Access Control Markup Language. OASIS committee, 1.1 edition, Aug 2003. Committee specification. 1.2 1.3 2.2.3.1


[Jav03] JavaCC project group. JavaCC project, Aug 2003. Information about JavaCC and jTree can be found at [https://javacc.dev.java.net/](https://javacc.dev.java.net/)


[KMPP01] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. In SACMAT’01, Chantilly, Virginia, USA, May 2001. ACM Press. 2.2.3.1


[LGF03] Ninghui Li, Benjamin N. Grosf, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorisation. In ACM Transactions on Information and System Security (TISSEC), volume 6, pages 128–171. ACM Press, Feb 2003. 2.2.1.2.2.3.2.2.4.3.2.2.4.3


[Li00] Ninghui Li. Delegation Logic: A Logic-based Approach to Distributed Authorisation. PhD thesis, New York University, Sep 2000. 2.2.4.3


[PS03] Calvin Powers and Matthias Schunert. Enterprise privacy authorisation language (EPAL 1.2). W3C Member Submission, Nov 2003. Information about this language can be found at http://www.w3.org/Submission/EPAL/.


[Som05] Fabio Somenzi. CUDD: Colorado University decision diagram package 2.4.1, May 2005.


[Sun03] Sun Microsystems. Sun’s XACML implementation, Aug 2003. Information about this implementation can be found at http://sunxacml.sourceforge.net/


[ZRG05b] Nan Zhang, Mark D. Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model-checking. This work has been submitted to the Journal of Computer Security, Dec 2005.