MODELLING AND VERIFYING DYNAMIC ACCESS CONTROL POLICIES USING KNOWLEDGE-BASED MODEL CHECKING

by

HASAN NAJIB YOUSIF QUNOO

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

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
March 2012

To my grandmother

Mariam Mohamad Qunoo

1932–2009

Abstract

The purpose of access control policies in computing is to guarantee that access to resources is solely restricted to legitimate users. This clarity of purpose does not make the design of these policies any easier. Today's systems are large in size, have many users with different roles and can be accessed from anywhere and at any time. Systems often allowed users to perform actions and read data so that users can do their job. Such a reality made it inherently difficult to get access control policies right.

Recently, failure to get these policies right has resulted in breach of privacy laws, caused financial loss and threatened the integrity of critical systems. This thesis aims at providing and developing methods and tools to address this issue.

The error-prone process of designing access control policies can be aided greatly by the right tools and methods. A modelling language that can express these complex policies and an automated framework to analyse those policies for security flaws can help mitigate this issue. It allows the policy designers to simplify and reason about the access control policy automatically.

This thesis advances the modelling and verification of access control policies by using automated knowledge-based symbolic model checking techniques. The key contributions of this thesis are threefold: firstly, a modelling language that expresses dynamic access control policies with compound actions that update multiple variables; secondly, a knowledge-based verification algorithm that verifies properties over an access control policy that has compound actions; and finally, an automated tool, called *X-Policy*, which

implements the algorithm.

This research enables us to model and verify access control policies for web-based collaborative systems. We model and analyse a number of conference management systems and their security properties. We propose the appropriate modifications to rectify the policies when possible. Ultimately, this research will allow us to model and verify more systems and help avoid the current situation.

Acknowledgements

I feel grateful to my supervisor Prof Mark Ryan who was a great help and provided timely advice and guidance throughout the development process and the writing up of this thesis. Prof Ryan was a remarkable inspiration by sharing his thoughts on how the project could be pushed further and improved upon. I was inspired by his attention to detail, passion for information security, healthy scepticism and never ending hunger for success. I believe for that I am a better scientist and researcher.

I also would like to thank Dr Eike Ritter, Dr Tom Chothia and Dr Georgios Theodoropoulos who have given time and thoughts on this thesis development as my Thesis Group. I would like to thank Dr Ritter for his quick thinking, formal rigour and his always two steps ahead technical insight, Dr Chothia for his encouragement and support and emphasis on the task at hand and Dr Theodoropoulos for his never outdated advice that I should never lose focus. That's why I am forever indebted to them.

It has been an honour for me to be a member of Birmingham's Formal Verification and Security Group and its always exciting Reading Group. The group members and its weekly meetings have been a great source of feedback. I am particularly grateful to: Myrto Arapinis, Tom Chothia, Eike Ritter, Masoud Koleini, Ben Smyth and Mark Ryan. I also benefited from discussions with Moritez Becker, Charles Gretton, Ben Jones and Juhan Ernits. Thanks to Dimitar P. Guelev; for lending me time with his great knowledge of math and logic.

A great deal of thanks goes to my office mates for the last four years: Damien Duff,

Jay Young, Zeyn Saigol, Ismail Bhula, Noel Welsh and Seyyed Shah. They made the daily grind of being a research student so much fun. They will be missed and I wish them all a very happy and successful life and career.

Damien Duff has a great gift. You can always get him to think about a problem he never heard of and within few minutes he will be able to ask questions and summarise it accurately. This is quite an impressive talent that I admire. Noel and Seyyed are always ready to get their hands dirty with code, Linux packages and never resist a bit of gossip in between.

Thanks goes to Steve Vickers, my research tutor. During the dark days when I was drifting into a research limbo, he helped me put things into perspective and let me see the big picture. He saw the researcher in me and helped me see it myself. I am certain if it was not for him, I quite possibly could have joined the ever exclusive club of Ex-Ph.D-candidates who never complete their thesis.

My family has always played a great part in my life specially during these years. My grandmother never had the privilege of formal education but that never prevented her from learning. She taught me that knowledge is precious and that everything is possible if you really work hard for it. My dad is my best friend and greatest hero. My Mum, you inspire me a great deal, your dedication makes every thing else pale in comparison. My brothers and sisters: Mohammad, Osama, Mohanad, Rofida and Riham; I am blessed to have you. I owe my deepest gratitude to you all.

The sweetest thanks goes to my beloved wife Farah, for being the rock of my life and the constant in all the variables that we had together in the last dynamic years. Thanks also to my kids: Najib and Joanne. Their smile has always replaced the burden of the toughest day with the joy of seeing them.

Thanks to all of you. I will remain indebted for you forever.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Research Motivation

Social forces are changing the role of computing in our society. Wikipedia, a crowd-sourcing encyclopaedia, is the largest of its kind [41]. Wikileaks, a crowd-sourcing-inspired whistle blowing website, was behind the biggest leak in military history [24, 84, 81, 86]. Social networks like Facebook and Twitter have been the catalyst for wide-spread multi-nation revolutions [48]. Access to collaborative document development tools like Google docs has been the subject of high-profile international and corporation conflicts [60]. Cloud-based conference management systems like EasyChair and EDAS are changing the way we manage academic research and conferences [95]. Our society as a result is increasingly dependent on these systems for their transformational empowerment, efficiency and accessibility. Thus, we are sharing our personal information and trusting computer systems with them like never before. Failure to meet the security objectives of software systems increasingly result in grave dangers, to individuals, corporations and societies at large.

Replacing paper-based systems with software systems has introduced new security

challenges. Software systems do not enjoy the built-in social control mechanisms that paper-based systems have. The accessibility of software systems makes them vulnerable to automated attacks. The portability of those systems' data makes them a valuable target. Security breaches often result in global effects on the system's end users. It comes as no surprise then, that designing efficient and secure software systems remains a daunting task

What is it that makes the development of secure software systems a challenging task? First, software engineers almost solely focus on the end-user features aspect of the software system without developing the proper software security requirement. Second, developers tend not to consider the full extent of any attacker's knowledge, imagination, skills and abilities. Third, there is a lack of tools and conventions that can help developers model, define and assess non-functional security requirements for software systems. We therefore need to provide software developers with the right languages, theories and tools to bridge the gap.

Taken together, these issues illustrate an interesting and current research problem in security engineering: that of dynamic access control policies in web-based collaborative systems; Social networking websites, conference management systems, collaborative document development tools, and application processing systems are all examples of web-based access control systems. Web-based collaborative systems are central systems that give users the ability to create and control access to their data. Access to data in these systems is dynamic; it depends on the state of the system and its configuration. Users with the right permissions and in the right system state can acquire information about the system state or execute actions that causes the system state to evolve into another state.

The size and the complexity of the system policy makes it difficult to analyse its security and correctness properties by hand. These systems are designed to preserve

the system integrity and serve their desired purpose. Systems might not always succeed; users can circumvent the system to gain illegitimate access usually by interactions of rules, co-operations between agents and multi-step actions.

The more accurate and precise a modelling language is, the more it can capture and analyse the real behaviour of the system. Developing an expressive and powerful access control policy verification framework that can capture large and intricate systems will help us understand policies, verify the fitness of the policies and discover possible security holes in the system. Fitness of policy in this context is defined by the ability of individual policy rules working together to preserve the system integrity against attacks that use interactions of rules, co-operations between agents and multi-step actions. Security holes such as allowing a reviewer to read another reviewer's review of a certain paper before she submits her own review or an author to review her own paper as explained in [121, 91]. Managers and decision makers at the moment are prevented from reasoning effectively about the actual access control process. The traditional way of using Access Control Lists (ACLs), XML-based policy files, or machine readable code for access control obstructs effective reasoning about access control policies at the enterprise scale. This process also requires a high level of technical skill.

It is also evidently important in the case of multi-agent collaborative systems to reason about user's knowledge about the state of the system. It allows us to model the user's allowed behaviour. As the access to the system is dynamic and depends on its state, the attacker needs to gather information about the system to be able to evaluate whether or not she can perform a certain action. Such a method allows the attacker to avoid being logged and/or flagged by the system monitor/administrator in the case of requesting prohibited information. Note that we consider attacker(s) to be the legitimate (authenticated) users of the system and the coalitions to be sets of authenticated users. Our method also allows us to reason about situations in which attackers form a coalition

where they share knowledge about the system. One of the benefit of knowledge-based reasoning is that the produced *strategy* derive the system to undesirable states without allowing situations where the user cannot backtrack from or unintentionally destroy a certain piece of information while carrying on the strategy.

Most academic conferences are managed using software systems. These software systems allow authors to submit their papers and the PC chair to manage users and download, assign and evaluate these papers with the other PC members. The system permissions are set based on the combination between the users roles, system configurations and state. Thus, the access to the data is dynamic, complex and hard to understand. It also important to ensure the integrity of the system. One might want to verify properties like "a single user cannot review the same paper twice". or "an author cannot review her own paper". It is also desirable that in the case where the system policy fails the property, the tool outputs a counterexample strategy to help the policy designers to identify and fix the problem.

The main thesis of this research is that a knowledge-based model checking verification framework can be developed to model and verify dynamic access control policies for real-life web-based collaborative systems automatically.

We propose *X-Policy*, a knowledge-based verification tool for dynamic access control policies. *X-Policy*'s modelling language with its corresponding query language and verification algorithm are based on concepts of *RW* [121], but extends it with the ability to express and analyse compound actions. *X-Policy*'s modelling language allows us to describe access control system as two sets of rules: *read permission rules* and *action execution rules*. A *read permission rule* allows us to specify the permission conditions that a user needs to satisfy to be able to read the value of a certain system variable. An *action execution rule* allow us to specify the system operation and its execution permissions where the user can change one or more of the system variables as a compound action (all

or none). We assume that the user has knowledge of a system variable if she or any other member of her coalition has previously read that variable. Users can share information with the coalition as they can communicate with each other using an outside channel. We also assume that the user will maintain that knowledge of the variable if the user has performed an action that changed the value of that variable, and that the coalition has an exclusive access to the system. While carrying out the strategy, the agents are assumed to completely know the policy of the system. 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 she could know. Therefore the attacker is aware of the system policy while executing the attack (strategy). While such assumptions give the attackers what seem as an unfair advantage, it is certainly desired that the system be tested in the most rigorous way. It also covers scenarios where attackers automate their attack by simulating access requests to the system. Such attacks are common in web-based collaborative systems and very important to model.

One of the most important features of *X-Policy* is its ability to express compound actions that update multiple variables. This is crucial for modelling collaborative systems. For example, when a user responds to an invitation to take a certain role, the system updates the status of that invitation and then update the user's role according to her answer. Similarly, once a user account is deleted from the system all the user's roles will be consequently deactivated.

X-Policy implements a knowledge-based algorithm where a user can input a system policy specification and a property (query). *X-Policy* then verifies whether or not the system satisfies the properties. If the system fails, it outputs a strategy that shows how the attacker or the coalition of attackers can achieve the goal. The remainder of this chapter is structured as follows. A detailed description of the contribution made by this thesis will be presented in Section 1.2. A summary of publications will be presented in

Section 1.3 and finally, the thesis structure will be summarised in Section 1.4.

1.2 Our Contribution

We have explored, studied and classified the field of access control policies both in terms of the type of systems that are being targeted, expressiveness and analysis capability of these languages, frameworks and tools. We have developed *X-Policy*, a knowledge-based verification tool that can express and analyse dynamic access control policies for vulnerabilities. The tool is available at: `http://www.cs.bham.ac.uk/~hxq/xpolicy`. The *X-Policy* tool can be used to detect attacks where the attacker can act as a coalition of users, use the system, share knowledge and collaborate with each other to achieve the attack.

First, a modelling language that specifies dynamic access control policies. *X-Policy*'s modelling language can express: read permission where users read the modelled system variables and compound actions where the user can update multiple variables. The simple syntax of *X-Policy*'s modelling language allows us to build simple yet expressive models of the access control policies. Intentionally, the resulting models are easy to build and understand by policy developers.

Secondly, a knowledge-based verification algorithm that verifies properties over an access control policy that has compound actions. We extend the method used by *RW* [121] to maintain the integrity constraints of the system and to handle the system behaviour resulting of the execution of compound actions. The ability for this algorithm to factor user's knowledge allow us to reason about coalition of users and the ability of users to share knowledge of the system. We also provide the proof of correctness and complexity analysis of the algorithm

Third, a case study in which we analyse a number of security properties of ***EC*** model, which is based on EasyChair conference management system. we address the issue of the

lack of real life examples and case studies in the field of access control policies. We also provide a number of conventions that can be adopted to model other web-based collaborative systems and similar conference management system like HotCRP and iChair.

Finally, a software implementation of the model checking algorithm. We apply the software implementation *X-Policy* in analysing ***EC***. We present our results and discuss the possible solutions based on the generated strategy. We also strengthen our algorithm complexity analysis by comparing practical performance results with similar tools and we discuss our findings.

1.3 List of Publications

Most of this thesis have been published in a number of papers and reports as detailed below:

- Hasan Qunoo, Masoud Koleini, and Mark Ryan. Towards modelling and verifying dynamic access control policies for web-based collaborative systems. W3C Workshop on Access Control Application Scenarios, November 2009

This paper provides the contexts and the motivation behind *X-Policy*.

- Hasan Qunoo and Mark Ryan. Modelling dynamic access control policies for web-based collaborative systems. In Sara Foresti and Sushil Jajodia, editors, *DBSec*, volume 6166 of *Lecture Notes in Computer Science*, pages 295–302. Springer, 2010

- Hasan Qunoo and Mark Ryan. Modelling dynamic access control policies for web-based collaborative systems. Technical Report CSR-11-08, University of Birmingham, School of Computer Science, August 2011

This paper and technical report provides the content of Chapter 3 and 5.

- Hasan Qunoo. X-Policy: Knowledge-based verification tool for dynamic access control policies. Technical Report CSR-11-09, University of Birmingham, School of Computer Science, December 2011

This technical report provides the content of Chapter 4 and 3.

1.4 Thesis structure

The rest of this dissertation is structured as following:

- Chapter 2 review research related to access control policies, with particular focus on verification frameworks and the dynamic aspects of policies. We try to classify the research done in the field based on the type of systems and the different problems they attack. We provide gap analysis of the current literature in access control policies. We explore the limitations of current approaches. We use the appropriate examples to illustrate both the capability and limitations of each framework.

  It should be noted here that Chapter 2 serves as a scene-setting survey which we use to clarify the research problem we are address in this dissertation.

- Chapter 3 present the *X-Policy* modelling language. We begin by providing the language syntax and informal semantics. We illustrate how we can express reading and compound actions using *X-Policy*. We also introduce the query language and we provide a number of examples to illustrate the use of *X-Policy* query language to write security properties.

- The model checking algorithm is explained in details in Chapter 4. This includes arguments about correctness and complexity analysis.

- Chapter 5 presents the process of modelling and verifying the access control policy of EasyChair conference management system. We give some background information

of conference management systems and overview of the conventions used in the constructing the model. We then describe the access control policy in *X-Policy* and provide an analysis of number of security properties. We also suggest a number of changes to the access control policy when appropriate. We conclude this section by our observations and survey of security properties of similar conference manganese systems.

- Chapter 6 briefly describe our implementation of the tool. This includes the usage information and system requirements, a discussion of the *X-Policy* tool. It compares the performance, results and analysis of *X-Policy* framework against other similar tools as applicable.

- Chapter 7 concludes this thesis by a summary of main contributions, lesson learned and an outline of future work.

CHAPTER 2

Background and gap research

Access control theory has been the focus of an intensive research in the past decades. In the past few years, a mixture of theory and software engineering techniques has been developed. This chapter provides an introduction to the problem of modelling and verifying dynamic access control systems and a review of relevant work. After providing a background to the research problems in access control, we explore the various languages, systems and frameworks developed to solve these problems and we try to classify them based on their purpose, technique and expressibility. This chapter sets the scene from which the Chapters 3 and 4 extend the *RW* [121] framework and provides the link between them and the case studies discussed in Chapter 4, where we model and verify the access control policy of a number of web-based collaborative systems.

This Chapter proceeds as follows. Section 2.1 introduces and motivates the problem in access control with the introduction with the early models. Section 2.2 describes the problems and proposed solutions for access control in distributed systems. We then discuss the issue of dynamic access policy analysis and the various proposed solutions with a special focus on model checking based access control frameworks including *RW* in Section 2.3. We then summarise and classify these solutions based on their contexts and

try to set the scene for the following chapters in Section 2.4.

2.1 Access Control: Background

The process of *mediating* every *request* to resources and data maintained by a system and determining whether the request should be *granted* or *denied* is called *access control* [33].

Access control policy, the set of *regulations* that can be specified in an appropriate language and then enforced by the access control mechanism, needs to be defined properly. Access Control Policies allow us to reason about the security properties of these policy as a whole. This can facilitate, in the case of an efficient implementation, powerful enforcement of rules authorising access to resources.

2.1.1 Access Control Models

Access control has been a significant research interest. The control of access to files and resources, has appeared for long time as a crucial part of operating systems [49, 66]. Access Control Models like MAC [17], DAC [100, 50] and RBAC [99] has been developed to organise access to resources.

In the **mandatory access control (MAC)** model, first formalised by Bell and La-Padula [17], access to data is based firstly on the data as sensitivity or classification, represented by a security label, and secondly on the clearance of the user, represented also by a security label. Security labels forms a lattice. One common set of labels used is *unclassified* $\leqslant$ *confidential* $\leqslant$ *secret* $\leqslant$ *top-secret*. The systems, known as *multilevel security systems* using MAC are based two principles; First, all users can read information with classification no higher that the one they are granted. Second, no user can write to a lower classification. As MAC was designed for military systems, the administration is centralised and delegation is forbidden. Such specification does not suit well other purposes, especially commercial organisations due to its inflexibility.

Discretionary access control model (DAC), less constrained model, where the owner of an object is trusted to manage access permissions to other users of the system. Users also can delegate control over an object to another user [100, 50]. An Access Control List (ACL) attached to each file is a classical approach used in operating systems like Windows, Linux and Unix to restrict users' access to the file system. Permissions are specified in *rwx* triple for the owner, group and others specify the rights to **r**ead, **w**rite, or e**x**ecute the file. Figure 2.1 shows the ACL of a file which belongs to Bob, who is a member of the group Staff. Access control matrices [49, 66] will be stored in columns. The DAC model is hard to administer as organisations grow. Any change to the policy can be an expensive and tedious process.

r w x r - x - - x Bob Staff

Figure 2.1: An access control list.

To fill the gap between MAC and DAC, **role-based access control (RBAC)** has been proposed to compromise between MAC, which is too rigid, and DAC, which is hard to administer.

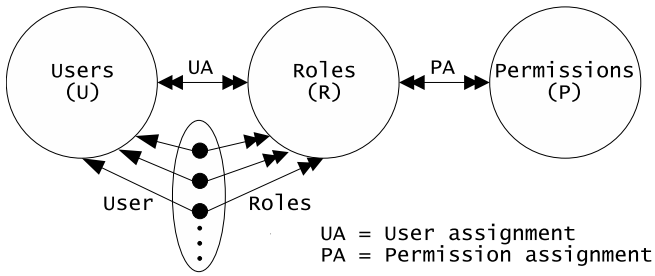

Figure 2.2: The $RBAC_0$ model.

As in real organisations, users are assigned roles or job functions. each role is associated with access permissions. Many different RBAC models have been proposed over the past few years but $RBAC_0$ [99] is often considered the core model. If we consider the set of

users (u), roles (r) and permissions (p) as key components, the $RBAC_0$ policy can be expressed as user assignment (UA) relations with roles and permission assignments (PA) to roles as shown in Figure 2.2. $RBAC_1$ has extended $RBAC_0$ with role hierarchies, within senior roles inherit permissions held by junior roles. $RBAC_2$ has introduced constraints on relations to $RBAC_0$. $RBAC_3$ integrates constraints and role-hierarchies.

Other models has been proposed in the area of access control [39, 85, 83, 30]. [39] defines a language for specifying authorization and obligation policies of an intelligent agent acting in a changing environment. However, it does not allow the definition of actions and their side effects. UCON [85] propose a conceptual framework that covers traditional access control, trust management, or DRM areas in a systematic manner to provide a general-purpose, unified framework for protecting digital resources. However, UCON proposed framework is orthogonal to the scope of this thesis which is verifying dynamic access control policies. [83] proposes the use of D-algebra for composing access control policy decisions while [30] present a formal, logical framework for the representation and analysis of authorization and obligation policies. Halpern and Weissman [47] have demonstrated how a fragment of first-order logic can be used to represent and reason about access control policies. Barker [8] proposed Status-Based Access Control which is more expressive than RBAC. These proposals differ from *X-Policy*; *X-Policy* focuses on expressing and analysing access control policies with atomic actions using automated model checking analysis using agent's knowledge while these frameworks are concerned with the specification of authorisation and obligation policies.

2.2 Access Control in Distributed Systems

As systems expand, the concept of systems evolve as well to include large-scale, heterogeneous and decentralised systems; and so do the access control concepts.

2.2.1 Authentication and Authorisation in distributed Systems

The way the access control systems were first developed did not consider today's environments. Today millions people all over the world are online, with whose introduction a new set of challenges is introduced [65]: *attack from any where, sharing with anyone, automated infection,* and *hostile code, environment, and hosts.* These factors make the design and analysis of system security a complex task.

Access control theory and practice provides mechanism to implement security in the modern systems. Figure 2.3 shows Lampson's model. The guard needs to know two piece of informations to reply to a request:

Authentication information which identifies the principal who made the request. This information often consist of some statements digitally signed to ensure their integrity and authenticity; these are called *credentials* [70].

Authorization information which says who is allowed to do what to the object. This information varies from an access model to another as we can see in section 2.1.1.

Separating the guard from the object makes reasoning about the access control policies simpler [2, 65, 67]. Even so, access control seems hard to get right.

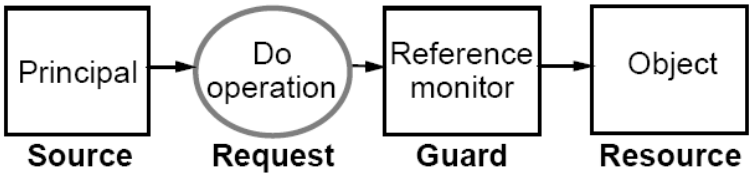

Figure 2.3: Lampson's access-control model.[67]

Over the years, many theories and systems for access control systems were developed. Many logics and calculus like [1, 9, 2] were proposed. The main approach is to study the concepts, protocols and algorithms of access control system and develop a ***notation***

for representing *principals* and *their statements* [1, 2]. Fournet [38] uses Spi calculus and applies it to a Conference Management System to check whether a distributed implementation based on communication channels and cryptography complies with a logical authorization policy. Swamy [109] which presents FINE, a new source-level security-typed language, focuses on the enforcement of dynamic security policies. However, the FINE policy language is an ML-level language and is concerned with the enforcement of the policy rather than analysing security properties. FINE also does not express high-level actions like the one considered by *X-Policy*.

2.2.2 Decentralised Trust Management

In recent years, there have been many proposals to develop a high-level access control policy language with the appropriate mechanisms to address the problems of decentralised administration, unified expressive and abstract languages to express policies, credentials, and relationships collectively. Recent researchers refer to these problems collectively as the ***"trust management problem"*** which was first mentioned in [20].

Soon after early proposals to solve trust management problems like PolicyMaker [20] and KeyNote [19], the complexity of analysing the web of trust has produced the need for a more attractive foundation. DeTreville was first to propose Binder [32], a Datalog based security language. Since then Datalog has become the foundation of recent trust management systems. Researchers are mainly attracted to Datalog [77] as they can start from a tractable and expressive language with the advantage of deducing trust relations effectively based on well developed *logic programming concepts and deductive databases* [21]. In most of those languages, we express policy rules in the form $\mathsf{if}\langle\mathsf{conditions}\rangle\mathsf{then}\langle\mathsf{goal}\rangle$. Note that while QCM [43] and its successor SD3 [58] are earlier than Binder in using Datalog, they focus on building secure DNS servers, public key directory and distributed repository. SD3 is the first trust management system to consider automated credential

retrieval. However, the fact that SD3 does not have an access control decision engine and focuses on the law level handling of the certificate retrieval, highly level authorisation languages like Cassandra and SecPAL. Cassandra and SecPAL seem to be influenced more by Binder and RT as [11], for example, classifies it with KeyNote and PolicyMaker as a low-level trust management system.

Recent languages like the RT family [70, 73, 72, 71, 69], Cassandra [16, 11, 15], SecPAL [13, 10, 34] have developed methods and solutions to high level authorisation problems problems, by adopting concepts mainly from Programming with Constraints [76] or Deductive Database [114], to improve the expressiveness of the languages and maintain their tractability. Such an expressiveness has enabled researchers like Moritz Becker [16] to formalise and express the UK National Health Service's National Programme for Information Technology (NPfIT) policy in the Cassandra framework.

The ***trust management problem*** refers to decentralised administration, unified expressive and abstract language to express policies, credentials, and relationships collectively.

Services available on the network are accessed from everywhere where resource owner and requester do not know each other. Therefore, there is a need to have a trust management framework to maintain the security of the system by ensuring that access to resources is restricted to legitimate users. Trust management framework must accommodate *appropriate notions* of *users security policies*, their *credentials* and their *complex trust relationship* where each service in the system has to maintain *locally* its *own belief and notion of trust.*

Many languages like Binder [32], PolicyMaker [20] and KeyNote [20], SPKI [7], SDSI and RT [72], XrML, Cassandra [16] and SecPAL [13] are examples of trust management languages. We can classify those proposals by their ideas and techniques. Noticeably, recent languages are based on ideas and techniques from logic programming. As a result,

those languages have inherited logic programming features like:

- ***Negation as failure:*** This happens due to the closed world assumption which is the presumption that what is not currently known to be true is false; i.e. if we can not prove a positive literal in the role conditions to be true then its false.
- ***Function symbol free:*** Datalog is function symbol free which gives tractability. On way to overcome such a limitation is by using Datalog with constraints (Datalog^C) [70].
- ***Quantifiers and roles negation:*** Datalog does not allow negation or quantifiers on roles; for example we can not express roles like:
  $\forall x(\text{Permitted } (x, \mathsf{apply\text{-}for\text{-}driving\text{-}license()}) \longleftarrow \neg\mathsf{Underage(x)})$.

We highlight those frameworks in the following sections and we discuss the concepts and methods they use.

2.2.2.1 PolicyMaker and KeyNote

PolicyMaker [20] is a service that acts as part of the resource guard or as an independent daemon. It inputs the request, the collection of credential and the local policy rules and outputs a the answer to the request as we can see in the Figure 2.4. The process of answering the question "Do the access rules and credentials authorize the request?" is called "proof-of-compliance". PolicyMaker provides a simple language to specify trusted actions and relationships. The PolicyMaker language enables two operations: *assertion* and *query* in the following form:

```
Assertion : source ASSERTS authstruct WHERE filter ;
Query     : keylist REQUESTS string condition ;
```

We might like to set a policy rule: trust Daffy (whose key is pgp: "4D DA 09 36 73 1D C5 33 BB 93 EB 4B 35 02 24 7E") to act as an agent on behalf of BigCo.:

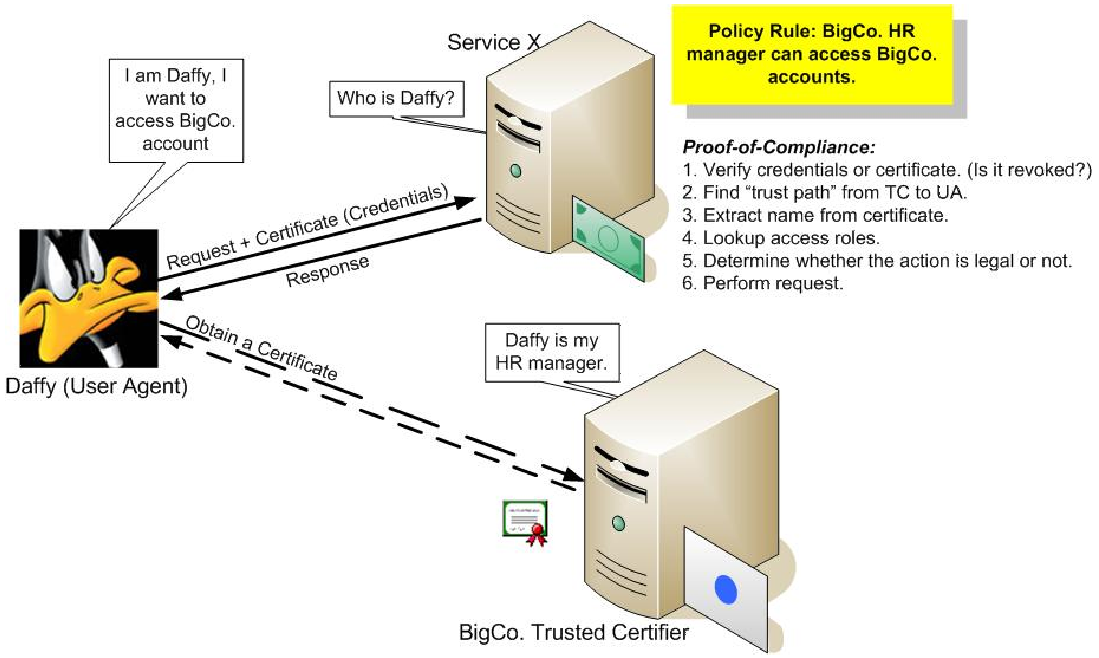

Figure 2.4: A trust management scenario

```
Policy ASSERTS

pgp: "4DDA0936731DC533BB93EB4B3502247E"

WHERE

 PREDICATE=regexp:"Organisation: BigCo.",
```

As you can see, the entities are identified by there public key. Then we can write the query from Daffy:

```
pgp: "4DDA0936731DC533BB93EB4B3502247E"

REQUESTS "From Daffy

          Organisation: BigCo.";
```

KeyNote, PolicyMaker's successor, shares the same semantics and spirit of PolicyMaker with small differences. KeyNote act as an fully programmable policy language written in C-like expressions. KeyNote responds to requests with "accept" or "deny" and is devised with a credential signature verification algorithm built in it. PolicyMaker and KeyNote

has a limited expressiveness and evaluation power as it is focused on the handling of public keys management rules that fail to express high-level policies with user roles and permissions.

2.2.3 Logic-based Access Control Policy Languages

2.2.3.1 RT

RT family [70, 73, 72, 71, 69] of attribute-based access control (ABAC) languages where authorisation in collaborative environments should be based on authenticated attributes of the entity was first proposed by Li, Mitchell and Winsborough. It uses Delegation Logic (LP) proposed by [69], a monotonic logic that extends Datalog.

For example we can express the following conference management concept to its equivalent RT rules:

- Chair asserts that Alice is a PC member:

  Assertion: Chair says isPCmember(Alice).

- Chair trusts any PC member to assign a subreviewer:

  Delegation: X delegates isSubreviewer(?Y) to Z
  if Chair says isPCmember(?X).

  Where isSubreviewer(?Y) means that any arbitrary entity Y can become Subreviewer if an entity X assigned Y to be become a subreviewer, given that X is a PC member.

- A PC chair requires a threshold of 3 positive reviews of a paper to be able to make a decision as to whether to accept or reject that paper. This rule can allow the set of reviewers to be specified:

dynamically:	Chair delegates PaperApproved(?P)
	to threshold(3, ?X,Chair says isPCmember(?X).
or statically:	Chair delegates PaperApproved(?P)
	to threshold(3, [Alice,Bob,Eve]).

Where we specify the set of reviewers to be Alice, Bob and Eve. The threshold here means that the maximum number of approval of a papers is three. Where the first one allow any three PC members to approve the paper while in the second statement it has to be Alice,Bob and Eve who approve it.

Policy rules are called credentials in RT_1 [71]. In RT_1, we write A.R to express that the principal A controls the role term R. There are four statement types that can be used to define an RT_1 credential (translatable to Datalog):

$Type1 : A.R \longleftarrow B$	A says B is a member of role R which is translated to the delegation logic equivalent of: $isMember(B, A.R)$
$Type2 : A.R_0 \longleftarrow B.R_1$	A says X is a member of role R_0 if B says X is a member of role R_1 which is translated to the delegation logic equivalent of: $isMember(?X, A.R_0) \longleftarrow$ $isMember(?X, B.R_1)$
$Type3 : A.R_0 \longleftarrow A.R_1.R_2$	A says X is a member of role R_0 if a member of role R_1 Z says X is a member of role R_2 which is translated to the delegation logic equivalent of: $isMember(?X, A.R_0) \longleftarrow$ $isMember(?Z, A.R_1),$ $isMember(?X, ?Z.R_2)$
$Type4 : A.R_0 \longleftarrow A_1.R_1 \cap A_2.R_2 \cap ... \cap A_k.R_k$	A says X is a member of role R_0 if X is a member of $A_1.R_1 \cap A_2.R_2 \cap ... \cap A_k.R_k$ which is translated to the delegation logic equivalent of:: $isMember(?X, A.R_0) \longleftarrow$ $isMember(?X, A.R_1)$ ... $isMember(?X, ?A.R_2)$

where $R = r(h_1, ..., h_n)$ where $h_1, ..., h_n$ are the role attributes. RT_1^C [70] share the same semantics and logic of RT_1 but allows ***tree*** and ***discrete*** constraints domains on the roles (credentials) attributes. It translates its roles to Datalog with constraints: Datalog^C [70]. Datalog^C [70] uses a least fixed point algorithm to apply the constraints and deduce facts. Although those constrains contributed to it's expressiveness, It is considered limited as constraints on rules or entities are not allowed to maintain the language tractability. RT is stateless and cannot express the dynamic aspect of access control policies.

2.2.3.2 Cassandra

Motivated by NHS Electronic Health Records (EHR) [35], Cassandra [16, 11, 15] extends on the above trust management systems and languages. It proposes a framework where each entity in the network runs a Cassandra agent with a local policy to control access to its resources as a set of rules based on Datalog^C. Rules have the form:

$$p_0(\vec{e_0}) \longleftarrow loc_1 \diamond iss_1.p_1(\vec{e_1}), ..., loc_n \diamond iss_n.p_n(\vec{e_n}), c.$$

where loc_i and iss_i are the name of the entity which assert the predicate and its issuer, respectively. c is a constraint from constraint domain. $loc \diamond iss$ can be interpreted as loc *says* iss *says* $p(\vec{e})$. The predicate name p can refer to an arbitrary user-defined names or one of the six special predefined predicates:

1. $canActivate(e, r)$: expresses the fact that entity e can activate the role r.
2. $hasActivated(e, r)$: expresses the fact that entity e has activated the role r.
3. $permits(e, a)$: the entity e can perform the action a.
4. $canDeactivate(e_1, e_2, r)$: the entity e_1 has the power to de-activate the victim entity e_2 from the role r.
5. $isDeactivated(e, r)$: Cassandra supports cascade revocation where the deactivation of one role will trigger the deactivations of other (local) roles.

6. $canReqCred(e_1, e_2.p(\vec{e}))$ the entity e_1 is allowed to request credentials issued by the entity e_2 and asserting the predicate $p(\vec{e})$.

Cassandra also supports a number of constraints domains and aggregation rules to express negation like "entity has not activated a role r" or "no entity has activated a role r". The use of aggregation has the benefit of expressing grouping and cardinality constraints, e.g. a patient can not assign more than three agents at once. Cassandra is a **state aware** access control policy language which depend on past events, or the system state like: "a reviewer may access other reviewers' reviews for a paper if he is assigned to this paper and he has submitted his review of that paper" or "a reviewer can be assigned to review a paper if he is not the author of that paper". We can express such a policy in Cassandra as the following policy rules:

canActivate (chair, Assign-for-reviewing(pcmember, paper)) ⟵
    hasActivated(chair, Chair()),
    canActivate(pcmember, PCMember()),
    paper-author-regs(0, paper, pcmember),
    currenttime() > deadline.

Cassandra uses a SLG^C resolution [94, 111] which is a top-down, goal oriented evaluation algorithm for Datalog^C to evaluate queries. It also support cascaded revocation using $canDeactivate$ and $isDeactivated(e, r)$ request:

isDeactivated (pcmember , reviewer(paper)) ⟵
    isDeactivated(chair, assign-for-reviewing(pcmember, paper)).

2.2.3.3 SecPAL

Security **P**olicy **A**uthorization **L**anguage (SecPAL) [13, 10, 34] is stateless Datalog^C based decentralised **authorisation policy** language which focuses on **controlled delegation**

and **principal aliasing** which can not be expressed by Cassandra, RT and other trust management languages. However, It does not handle state-aware policies and revocations. SecPAL Authorisation policies are specified as sets of assertions of the form:

$$\text{A } says \text{ } fact \text{ if } fact_1,\ldots,fact_n,\text{c.}$$

Where A is the issuer; the $fact_i$ are the *conditional facts*; and c is the constraint. SecPAL does not place any restriction on the choice of constraints domain, as in Cassandra. Instead, it enforces syntactic assertion safety conditions to ensure that all facts are flat (ie. all $vars(fact)$ are safe and can be deduced from within the same assertion rule where a variable $x \in vars(fact)$ is safe iff $x \in vars(fact_1) \cup \cdots \cup vars(fact_n)$). SecPAL supports the following idioms:

- **Controlled delegation** SecPAL uses the primitive *"can say"* e.g. "Chair can delegate to Daffy the ability to say who is a PC member." can be expressed as: Chair *says* Daffy *can* say_D x is a PC member. say_D can be say_0 to prevent Daffy from re-delegate capability of saying who is the PC member which they call **"depth-bounded delegation"**, or say_∞ to indicate that the capability is re-delegate-able.

- **Principal aliasing** where we can use *"can act as"* to express role hierarchies like:

  BigConference *says* PCmember *can read* file://conference/forum/
  BigConference *says* PCmember *can act as* Subreviewer.
  BigConference *says* Chair *can act as* PCmember.
  BigConference *says* Taz *can act as* Chair.

  In this scenario, Taz can access file://conference/forum/.

- **Parametrised Attributes** to encode parametrised roles, attributes and privileges. SecPAL introduces verb phrases like `can write a review of [-]` and `is a reviewer of [-]`.

BigConference *says* x *can write a review* of paper if

x is a *is a reviewer of* paper

- **Domain Specific Constraints with safety conditions.** We can classify the constraint domains it support to:

  Basic Constraints which include linear constraints (e.g. numerical inequalities), hierarchical constraints (e.g. path constraints) and regular expression (e.g. email address filtering).

  Additional Constraints SecPAL supports negation with safety conditions mainly using groundness analysis to ensure the constraints tractability. Constraints with depth-bounded delegation can be combined to provide a **width-bounded delegation** rules like:

  Chair *says* Daffy *can say*$_D$ x is a PCmember if

  x possesses Email email,

  email matches *@*.edu.

  In that case Daffy can not delegate to any principal unless his email address belong to *.edu* domain.

SecPAL supports Attributes Based Delegation like in RT, Cassandra and Delegation logic where there is a need to assert subject as a student and threshold-constraint trust. The controlled delegation allows SecPAL to enforce depth-bounded and width-bounded delegation which is not expressible in other languages.

On the other hand, SecPAL does not have the ability to model state-aware polices which require the dynamic insertion and deletion of facts to the database. Although SecPAL support static revocation assertion, It does not allow nested or dynamic revocation.

With such limitations, we found it hard to model RBAC policies like the conference and EHR systems.

2.3 Analysing Dynamic Access Control Policies

Shafiq et. al. [105] uses a Petri-Net based framework for verifying the correctness of event-driven RBAC policies. Schaad and Moffett [102, 101] use Alloy [55, 53, 54, 56] to specify a RBAC models and separation of duty properties [97]. Model checking tools [78, 79, 25, 4] have the advantage of being more exhaustive than previous approaches. This alas can not be done using general purpose model checkers like SMV [78, 79, 25] and Mocha [4] as discussed in [119]. These model checkers can not capture the concepts of access control like permissions about permissions [98]. This has motivated the research towards more domain specific model-checking based frameworks [119]. The access control verification software suit Margrave [37] addresses the policy *change-impact* problem using Model-Checking techniques. Margrave allow the users to compute the changes between two policy files written in XACML and calculates the differences between the two policies.

RW [121, 122, 120] answers the question of whether there is a strategy for a coalition of agents to achieve a given goal given an access control policy written in the RW formalism. RW uses a fixed point algorithm [42] to model the agents knowledge of the state of the system. It extracts strategies which make the goal achievable.

2.3.1 RW

RW is a framework for evaluating and generating access control systems. It analyses and detects the security holes caused by ***interactions of rules, co-operations between agents*** and ***multi-step actions***. It argues that:

- While individual policy rules could be enforced properly, the using a combination of a number of these rules in a dynamic system can circumvent the policy rules and

still achieve an attack the system.

- An attack can be achieved by a set of agents through co-operation.

- An agent can achieve an attack using a sequence of actions.

RW propose a solution for those problems. RW's solution is a composite of the following steps:

1. Build a model ***M*** for the access control policy using the ***RW formalism***.

2. Express the model ***M*** in the machine-readable ***RW specification language***.

3. Specify a ***property*** of the framework to be verified. The property defines a goal, a coalition of agents and some conditions to serve as a pre-requirements for the checking to be performed.

4. If, as is ideal, the policy is immune to the malicious goals and guarantees the achievability of the legitimate goals, The framework implements a tool to convert the policy from the RW language into a policy file in XACML. In case the goal is achievable, RW , using a model checking algorithm, outputs a strategy the agent(s) may use to achieve the goal. The output strategy can be used to amend the policy.

Using RW syntax we can define an example access control system $S = \langle \Sigma, P, \mathtt{r}, \mathtt{w} \rangle$, where the set of propositional variables $P = \{u(p), x(p), y(p), z(p), t(p)\}$ and the set of agents $\Sigma = \{a\}$. For each $p \in P$, the read permission rule $\mathtt{r}(p, a)$ and the write permission rule $\mathtt{w}(p, a)$ is detailed in the RW formalism as following:

$$\mathtt{r}(u(p), a) \rightleftharpoons \bot \quad \mathtt{w}(u(p), a) \rightleftharpoons \bot$$
$$\mathtt{r}(x(p), a) \rightleftharpoons \top \quad \mathtt{w}(x(p), a) \rightleftharpoons \neg u(p)$$
$$\mathtt{r}(y(p), a) \rightleftharpoons \top \quad \mathtt{w}(y(p), a) \rightleftharpoons u(p)$$
$$\mathtt{r}(z(p), a) \rightleftharpoons \top \quad \mathtt{w}(z(p), a) \rightleftharpoons x(p) \vee y(p)$$
$$\mathtt{r}(t(p), a) \rightleftharpoons \top \quad \mathtt{w}(t(p), a) \rightleftharpoons x(p)$$

To perform model-checking, a concrete instance based on the template needs to be constructed and a query has to be specified. We now can write a query to ask the question whether there is a strategy or guessing strategy for a to set z's value to false:

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

AcPeg (Access Control Policy Evaluator and Generator) is a Java implementation for RW framework. We can use AcPeg to model-check access control policy written in RW specification language. AcPeg also translate from the RW to XACML. AcPeg has implemented a three-level abstraction mechanism to handle large cases with a promising computational performance. AcPeg's methodology of abstraction using Counter-Example-Guided Abstraction Refinement (CEGAR) [26] techniques has improved the computational performance by keeping the model as *compact* as possible.

2.3.1.1 Strategy vs. guessing strategy

A *strategy* is a sequence of actions by which a coalition A of agents achieves a goal [122]. In that mode, only the actions that the agent(s) in the coalition know they can perform can be considered. This covers the scenario when trials to perform an illegal actions may be recorded in a log file somewhere which will be audited. The agent(s), in that case, are not welling to take the risk. A *guessing strategy* is similar to a strategy, except that it is not required that the agent knows he can perform each action [122].In this case, the agent(s) are willing to take a risk that the action will be denied. The RW algorithm will model

the learning process for the coalition of agents to compute whether there is a strategy or guessing strategy to achieve goal. An example of a guessing strategies produced by *RW* is shown in Figure 2.1.

```
[p=1 a=1]
Acting agents: [1]
Guessing strategy: 1
if (z(1) is true) by 1 {
   set z(1) to false by 1;
   skip;
}else {
   skip;
}
```

Listing 2.1: An example of a guessing strategy found by *RW* for the XUYZ Example.

2.3.2 Margrave

Margrave [37] is a software suite to analyse role-based access-control policies. Margrave has two components:

Verification system: given an access control policy written in XACML language and a given property, Margrave's verification tool will determine whether the policy satisfies the property or not. In case the property is not satisfied counter examples will be yielded.

Change-Impact analysis system: given two policies, the system will list a set of changes and summarise the differences between the two policies.

Margrave can process an access control policy if it is written in a restricted subset of the XACML standard. Considering a unviersity access control policy for assigning and accessing grades, the policy cen be specified by Roles, Resources and Actions only. The desired users of Margrave, the administrators of the grading system, should be able to

use Margrave to verify their XACML policy against a property like *"there do not exist members of* ***Student*** *who can* ***Assign ExternalGrades****"*. The tool restricts the policy to cases that permit ***assigning external grades***, then check to see whether students are enabled in the restricted policy. Margrave will yield the requests that lead to the violation (counter-example) as an error report shown in Figure 2.5. In Figure 2.5, Line 13

```
1   > Pr_1: FAILED Pr_1
2    Counter-example:
3    1:/Resource, resource-class, ExternalGrades/
4    2:/Resource, resource-class, InternalGrades/
5    3:/Action, command, Assign/
6    4:/Action, command, View/
7    5:/Subject, role, Faculty/
8    6:/Action, command, Receive/
9    7:/Subject, role, Student/
10   8:/Subject, role, TA/
11   12345678
12   {
13   10100011
14   }
```

Figure 2.5: Margrave Error report

represents the set of requests that comprise the counter-example. To explain this output, lines 3-10 list the subjects, resources and actions mentioned in the policy, while line 11 indexes the counter-example information against this header. The 1s in line 11 indicate which subjects, resources and actions are present in the counter-example, while the 0s indicate absence.

To elaborate on that example, the administrator saves a backup of the XACML policy file and changes the policy to prevent the incidence. She tests the *assigning external grades* property again and it passes. Now, she wants to check what impact that change has made. Margrave consumes the two policy files and analyses, without specifying any property, the differences between the two policies, which is output as a report shows in Figure 2.6. Margrave is implemented atop of the CUDD package [107] and represents

```
===== Pol_3 vs Pol_4 =====
1:/Subject, role, Faculty/
2:/Subject, role, Student/
3:/Resource, resource-class, ExternalGrades/
4:/Resource, resource-class, InternalGrades/
5:/Action, command, Assign/
6:/Action, command, View/
7:/Action, command, Receive/
8:/Subject, role, TA/
12345678
 {
 00010101 N->P
 00011001 N->P
 00100101 N->P
 00101001 N->P
 01010101 N->P
 01011001 N->P
 01100101 N->P
 01101001 N->P
 }
query: #f
```

Figure 2.6: Margrave Change-Impact report

the XACML policy as MTBDD [51] *multi-terminal binary decision diagram*. Each rule is represented by a MTBDD then the MTBDD are combined using the parametrised general algorithm *Apply*. Environmental constraints, like *no faculty member is also a student* are represented as boolean conditions. Users can write queries and test cases, one per run, using DrScheme [36], a programming environment for scheme. The Margrave language does not allow defining new types. This restriction makes the verification for properties like, *"students can not write grades"* un-representable in its formalism. The soundness and completeness of Margrave, with Respect to the subset of XACML, is proved [40].

2.3.3 DyNPAL/SNP

DyNPAL [12] is a policy language and analysis method that specify dynamic authorisation policy. DyNPAL is based on transaction logic [23, 22] which allow DyNPAL to express

bulk assertion and retraction of authorisation facts to the system state. DyNPAL offers two analysis methods: a finite domain method using Fast Forward AI planner [52] and automated theorem proving method for checking policy invariants.

In DyNPAL, a system state is expressed as a set of positive ground atoms. An atom is a predicate assigned to the unsorted appropriate arty. Atoms that are not in the state set are negative as DyNPAL uses negation by failure. Policy rules in DyNPAL are specified in two steps:

- what pre-conditions the system state must satisfy to allow the execution of the action `ActionName`.

- what the effects of executing that action is. There are two types of effect: a positive effect in which a set of atoms can be added to the state in the form of +Atom and a negative effect in which an atom is retracted from the state in the form of −Atom

DyNPAL also can express intermediate conditional effect in the form of $\{\pm\text{Atom}_i\text{: C}_j\}$ which mean add or remove the atoms in the form Atom_i such that C_j holds. In the context of conference management systems we can express the rule that a review can be read if the user is the user is a PC chair or if she is a PC member who is a reviewer of that paper and has already submitted her review. This rule is encoded in the following statements:

canReadReview(p, a, b, u)⟵
    chair(u)
canReadReview(p, a, b, u)⟵
    pcmember(u), reviewer(p,u),$\exists c$(submittedReview(p,u,c))

The head canReadReview can then be used in constructing other rules. Note that a and b are constants that represent agents of the system. We can also express the rule that an

agent can write a review of a paper if the agent is a PC chair or a PC member who is assigned to review that paper. We can express this as following:

canSubmitReview(p, a, b, u)⟵
    chair(u),+submittedReview(p, a, b)
canSubmitReview(p, a, b, u)⟵
    pcmember(u), reviewer(p,u),+submittedReview(p, a, b)

DyNPAL is a minimalistic language. It lacks the ability to establish the association between the action and the agent who executed that action and it does not handle the readability of variables. One the other hand DyNPAL reachability analysis algorithm considers a single path between two individual states an initial and a final state. A complimentary theorem proving method which check safety invariants based on first-order logic.

SMP [14], DyNPAL precursor, share the aim of DyNPAL but is less expressive as it does not allow the use of quantified variables in the body of the rules or conditional bulk update.

2.4 Setting the Scene (or Gap Analysis)

Most frameworks and systems agree on the importance of *roles* as the crucial concept to get an appropriate level of abstraction in defining access control policies. But there are several open issues, including:

Issues in modelling access control:

- Centralised vs de-centralised. Web-based access control systems (e.g. Facebook or EasyChair) are centralised, in the sense that everything happens on the server (we can ignore the fact that the client is remote). But some researchers are interested in inherently decentralised systems, in which there are many geographically distributed

decision points and a request can involve several of them. Protocols can be used to coordinate the access control decision.

- State-based (dynamic) vs stateless. If updates to the regular data maintained by a system can affect access control decisions, then we can consider it state-based (i.e., decision depends on state), or dynamic (i.e., decision changes with state). Facebook access control is of that type. If there is not a dependency, then it's stateless. An operating system's file system is of that type.

We can classify these systems as following:

Datalog-based frameworks (stateless / distributed) Datalog is a popular formalism for writing access control rules DeTreville was the first to propose a Datalog based security language called Binder [32]. Then there was the RT family [71]; and later, Microsoft's SecPAL [10]. Researchers are mainly attracted to Datalog [77] as they can start from a tractable and expressive language with the advantage of deducing trust relations effectively based on well developed logic programming concepts and deductive databases. Unfortunately, access control policies defined using Datalog are stateless, although Cassandra [15], a Datalog-based language, has a separate mechanism to maintain the authorisation state by inserting and retracting "hasActivated" facts according to the policy rules.

Decentralised authorisation languages (stateless / distributed) Gurevich et. al. introduced Distributed Knowledge Authorisation Language DKAL [44] and DKAL2 [45] that extend SecPAL's expressiveness (by allowing functions that can be nested and mixed). The access control verification software suite Margrave [37] addresses the policy change-impact problem using Model-Checking techniques to compute the changes between two policy files written in XACML. Cassandra, Margrave, SecPAL, DKAL, DKAL2 and other authorisation languages lack the ability to express the

dynamic aspect of access control where policies depend on and update the system state like those we have in real-life web-based collaborative systems like EasyChair, Facebook and LinkedIn. They, also, cannot express the effect of actions as part of the language and it has to be hard-coded in an ad-hoc way.

Dynamic authorisation languages (dynamic / towards distributed) More recently, SMP [14] and its successor DyNPAL [12] aim to specify dynamic policies with the ability to specify the effect of executing these actions. DyNPAL allows conditional bulk insertion and retraction of authorisation facts with transactional execution semantics (either all or none are committed). However, DyNPAL's declarative nature and minimalistic approach make it hard to follow the control flow of the actions. Also the lack of parameter typing in DyNPAL does not allow us to establish the relation between the agent who can execute an action and the action itself. SMP and DyNPAL also tend to focus on answering the question "under what conditions can an action be executed?" rather than "under what conditions can an agent execute an action?" which is indeed necessary to enable us to define agent coalitions and establish which agent is executing an action. It allows us to detect attacks where we are interested in who can execute a set of actions rather than whether a set of actions can be executed regardless of the actors involved.

RW-based systems (dynamic; centralised)

RW framework [120] can analyse the consequences of multi-agent multi-step actions by performing temporal reasoning. *RW* is a model checking based framework. However, RW does not allow us to express actions with multiple assignments needed to preserve the integrity constraints of the modelled system.

X-Policy extends the the modelling language and algorithm of RW to express and verify dynamic access control policies with compound actions with multiple assignments.

2.5 Chapter Summary

This chapter provides a structured review of access control theories and techniques. Access control models, logics and calculus for access control, decentralised trust management languages, dynamic policy analysis and verification frameworks are discussed. In the following Chapter we introduce *X-Policy* language.

CHAPTER 3

X-Policy : Modelling Language

The purpose of this thesis is to develop a knowledge-based model checking verification framework to model and verify dynamic access control policies for real-life web-based collaborative systems automatically. To fulfil this purpose we specify in this chapter the *X-Policy* modelling language.

This chapter is structured as follows: firstly we introduce the *X-Policy* modelling language. In Section 3.1, we begin by discussing the scope and design decision for *X-Policy* modelling language. We overview the *X-Policy* modelling language is Section 3.2. We then provide the syntax and the informal semantics of the modelling language of *X-Policy*. Finally, we summarise this chapter.

3.1 Scope, Design Decisions and Discussion

The *X-Policy* modelling language uses propositional variables to represent data, relations between data and permissions in an access control system. Rules in the access control policy adopted by the system are expressed by logical formulas built from the defined variables.

We specify system operations as *X-Policy* write actions or read permissions. *Write*

actions changes the state of the system. A write action in *X-Policy* can not read and change the state of the system at the same time. Although this is formally a restriction, most actions in collaborative web-based systems are indeed either a read or write and rarely both. This is true for EasyChair in particular. We believe that this is a sensible heuristic for modelling web-based systems. Users are only enabled to perform only one operation per time. A *read permission* statement allows the user to know the value of a ground proposition by returning the value of that proposition to the user who executed the program.

Flexibility and simplicity. *X-Policy* is a simplistic language. It has the minimum number of constructs while at the same time it is flexible enough to allows us to express the behaviour of large and complex systems. Indeed, *X-Policy* has no predefined predicates or action names. This allows the users the freedom to model generic systems.

Fine-grained conditional rules. *X-Policy* allows us to express access conditions as propositional logic formulae using agent's roles, system states and system configurations. We choose to express the access control policy as a set of allowance conditions. We also group the set of conditions an agent need satisfy to execute an action or read a variable in a single statement to avoid redundancy and to help us express the policy in terms of the system capabilities.

State-fullness and compound actions. In modelling access control policy, *X-Policy* allow us to express permissions using the system state variables but at the same time it allow us to express compound actions. These two properties allow *X-Policy* to express two kinds of situations: permission about permissions and correcting integrity constraints violations.

- **Integrity constraints** play a crucial role in expressing access control polices.

Neglecting integrity constraints, as *RW* does, affects its ability to capture the real behaviour of the model system and may lead in some cases to produce spurious strategies for the investigated properties. For example, when deleting a reviewing assignment, all sub-reviewing requests should be deleted. However, if that did not happen then a spurious strategy where a sub-reviewer is still assigned to that paper can be produced. In addition, integrity constraints increase the language expressiveness to capture essential concepts like: **mutual exclusivity**, **inheritance,** and **atomic actions**.

- **Permissions about permissions** is an important aspect of dynamic access control systems. When an agent executes an action, he may change the other agent's permissions. In the example of conference management system, a chair can execute an action to assign another agent Alice to the role PC-member. Consequently, Alice has gained access to the system as a PC member.

Readability and information flow Web-based collaborative systems generally provide information to users using their web interface. For example, a GP doctor should be able to find out the height and weight of her patient. *X-Policy* allow us to model the read permissions and the process of learning that the user can go through to enquire about the system state.

In the following sections, we will illustrate the various aspects of the *X-Policy* language with examples when appropriate. In this chapter we provide the language syntax and how a model is constructed from the system using informal semantics and examples which allow us to use this Chapter as a tutorial for the *X-Policy* language. However, we define formally when the query is satisfied in Chapter 4 in the form of the model checking algorithm and how we construct the knowledge states as in Sections 4.2 and 4.3.

3.2 *X-Policy* language specification

An *X-Policy* model consists of three parts: Access Control System, Access Control Model and Security Property.

Access Control System. In this part, we define three components:

System types. These are the types of the subjects and objects in the modelled system. In the example of conference system they can be Papers and Agents.

System predicates. These predicates could represent the role relationships, association between subjects and agents or system decisions. These predicates will be used to form ground propositions[1].

Read permission statements. Each statement specifies the conditions an agent has to satisfy to read each of the ground propositions. A *read action* allows the agent to know the value of a ground proposition by returning the value of that proposition to the user who executed the program.

Write action execution rules. These rules are used to specify the set of variable assignments for each action. These assignments detail the changes each action makes to the system state. We also specify the set of conditions an agent has to satisfy to execute each action. A *write action* allows the agent to change the value of a set of ground propositions using assignment statements. An action permission statement defines the conditions for an agent to execute an action. These conditions are defined as propositional logic formulae using the ground propositions and logical connectors.

Access Control Model Once the access control system is defined, we use it as a template to build a concrete model to perform the model checking analysis. This is

[1]We will explain this process of forming the ground propositions in the following sections.

done by specifying the model size which is defined by the number of individuals of each type.

Security Property We define security properties as a reachability query where we specify the initial state, goal state and the coalition agents. The tool will find out whether or not there is a strategy the coalition can derive to take the system from the initial state to the goal state.

3.2.1 Access Control System

An access control system S is defined as two sets of rules: *read permission rules* and *action execution rules*.

Let T be a set of types, which includes a special type `Agent` for agents, also let $Pred$ be a finite set of predicates. Each n-ary predicate has a signature $t_1 \times \cdots \times t_n \rightarrow \{\top, \bot\}$, where $t_i \in T$. For example, in the case of a conference review system, T can include Paper which we can define using type definition statement of the form:

$$\text{Type Paper;}$$

and $Pred$ can include the predicate

$$\mathsf{Author} : \mathsf{Agent} \times \mathsf{Paper} \rightarrow \{\top, \bot\}.$$

Note that each system will require a different list of predicates. We assume a set of variables V, each with a type. If $p \in Pred$ and $\overrightarrow{x} = x_1, \ldots, x_q$ is a sequence of variables of the appropriate type, then $p(\overrightarrow{x})$ is an *atomic formula*.

3.2.1.1 Read Permission rules

These definitions allow us to define whether or not a user has permission to access the truth value of an *atomic formula* and is of the form

$$p(\overrightarrow{y})\ \{\ \textbf{read:}\ \textit{formula};\}$$

where $p \in Pred$ and the variables in $\overrightarrow{y}$ occur in $\overrightarrow{x}$. *formula* defines the conditions for an agent `user`$\in$ `Agent` to read a system variable $p(\overrightarrow{y})$. *formula* is a logical formula which is defined using atomic formulae and logical connectors: $\neg$/`~` (negation), $\wedge$/`and` (conjunction), $\vee$/`or` (disjunction), $\rightarrow$ (implication), $\exists$/`E` and $\forall$/`A` (existential and universal quantification over variables of the appropriate type). The variables that occur in *formula* are required to be either in $\overrightarrow{x}$ or *user*. The *formula* defines the conditions for agents to read these variables as functions on its state.

3.2.1.2 Action execution rules

An action execution rule allows the user to change the truth values of an atomic formula and is of the form

$$\text{Action } \texttt{Actionname}(\overrightarrow{x}) \text{ :- } \{\texttt{writebody }\}\ \{\textit{formula}\}$$

where `writebody` is an expression formed from the following Backus−Naur Form (BNF):

$$\begin{aligned}\texttt{writebody} ::= \texttt{assignment} \mid \text{for } (v : t)\ \{\ \texttt{writebody}\ \} \\ \mid \texttt{writebody writebody}\end{aligned}$$

where v is a variable of the type t and an `assignment` is of the form $p(\overrightarrow{y}) := \top$; or $p(\overrightarrow{y}) := \bot$;. We allow an *atomic formula* $p(\overrightarrow{y})$ to occur at most once at the left of ":=" in an action to avoid ambiguity in computing the action effect. The `assignment` statements within the same action can be written in any order. All free variables in an

`assignment` must be declared either in a surrounding for-statement or in `Actionname` statement. Intuitively, a `for-statement` in an action is a 'macro' that is interpreted as multiple *assignment* statements. *formula* defines the conditions for an agent `user`$\in$ `Agent` to execute an action *act* $\in$ `Actions` where `Actions` is a finite set of all the actions in the defined system. The *formula* defines the conditions for agents to execute these actions as functions on its state.

3.2.2 Example: Expressing Dynamic Access Control Idioms

This Section expresses a number of access control idioms in *X-Policy*. The actions, variables and rules listed here are based on the ***EC*** model which Chapter 5 discusses in great detail. We define for a,b of type Agent, p of type Paper, P which includes:

Author(p,a)	Agent a is an author of paper p.
Chair(a)	Agent a is the chair of the PC.
PCmember(a)	Agent a is a PC member.
Reviewer(p,a)	Paper p is assigned to PC member a for reviewing.
View-sub-by-chair-permitted()	PC chairs can view the list of submissions.
View-sub-by-PCM-permitted()	PC members can view the list of submissions.

A **read permission rule** then can be expressed in the Listing 3.1 which expresses the permission rule for reading whether or not an agent a is a reviewer of a paper p which allows only an agent `user` who is a PCmember and is not the author of the paper p. Note that the agent `user` is a special agent that represents the user requesting access. **An action execution rule** also can be defined as in Listing 3.2, to assign an agent `a` the role of `PCmember`. Only an agent who satisfies the predicate `Chair` can execute such action. The Listing 3.3 expresses the action rule that only an agent who is acting as a chair can allow who can access the submissions: chairs only, PCmembers and chairs or no one. Using **compound atomic actions**, we can express a number of dynamic access

```
Reviewer(p, a)
 {
      read : PCmember(user)&~Author(p, user);
 }
```

Listing 3.1: An example of an read permission rule.

```
Action AddPCmember(a:Agent)
 {
      PCmember(a) :=true;
 }
 {
      Chair(user);
 }
```

Listing 3.2: An example action rule `AddPCmember` to assign an agent `a` the role of `PCmember`.

control idioms.

- *Mutual exclusion of roles:* for example we can express mutual exclusion between two roles. "A chair of a conference can not be a PCmember" is an example of such relation. We can specify this constraint in *X-Policy* in the form of the action rules detailed in Listings 3.4. Any PC chair can appoint or delete PC chairs. A PC chair can be demoted to become a PC member or the other way around. The chair cannot delete herself. This restriction is introduced to prevent the Chair from locking herself out of the system. Another example of such relation in the context of course modules can be "A student can not be a lecturer for the same module". Special care should be taken in specifying the initial state of the model so it starts from a state that satisfies the integrity constraints where no agent is PC chair and PC member at the same time. In addition it is important to take extra care when specifying the policy rules that other actions do not violate the role mutual exclusion

```
Action viewSubmissions2Chair ()
 {
      View-sub-by-chair-permitted():=T;
      View-sub-by-PCM-permitted():=F;
 }
 {
      Chair(user);
 }
Action viewSubmissions2All()
 {
      View-sub-by-chair-permitted():=T;
      View-sub-by-PCM-permitted():=T;
 }
 {
      Chair(user);
 }
Action viewSubmissions2None()
 {
      View-sub-by-chair-permitted():=F;
      View-sub-by-PCM-permitted():=F;
 }
 {
      Chair(user);
 }
```

Listing 3.3: Actions allowing the chair to control access to the submissions.

during the interaction of rules.

- *Role inheritance and hierarchy constraints:* for example "Once a PC member/Chair is deleted her assignments will be deleted." is expressed as the compound action `DeletePCmember` in Listing 3.5. When a user a is deleted, the variables `PCmember(a)` and `Chair(a)` will be changed to false. This also will change the value of all the variables `Reviewer(p,a)` to false for all the papers.

```
Action PromotePCmemberToPCchair(a)
 {
      PCmember(a) :=F ;
      Chair(a):=T;
 }
 {
      Chair(user) and  PCmember(a) and user!=a;
 }
Action DemotePCchairToPCmember(a)
 {
      PCmember(a) :=T ;
      Chair(a):=F;
 }
 {
      Chair(user)  and  Chair(a) and  user!=a;
 }
```

Listing 3.4: Actions that implement the mutual exclusion of roles: Chair and PCmemeber.

```
Action DeletePCmember(a)
 {
      PCmember(a) :=F;
      Chair(a):=F ;
      for (paper p:Paper)
           Reviewer(p,a)=F;
 }
 {
      Chair(user) and  PCmember(a) and user!=a;
 }
```

Listing 3.5: The compound action `DeletePCmember` ensures that all reviewing assignment of an agents are deleted once she is no longer a Chair or a PCmember.

3.2.3 Access Control Model

A system S described in the description part, as in section 3.2.1, is only a template. To perform model-checking, a concrete instance based on the template needs to be constructed. This task is done through a run-statement. The syntax of the run-statement

is:

`RunStatement` ::= run for `NumberTypePair` (“,” `NumberTypePair`) *

`NumberTypePair` ::= d `TypeName`

where d is an integer. When a run-statement is executed, the tool creates a model M which assigns each defined type t to a fixed number of elements σ_t as specified in the run-statement. We define $\sigma = \sigma_{t_1} \cup \cdots \cup \sigma_{t_n}$ as the set of all the individuals defined by M. We assume $\sigma_{t_1} \cap \sigma_{t_2} = \emptyset$ whenever t_1 and t_2 are distinct. These elements are then used to instantiate the relations defined by the parametrised predicates and actions. Once these two steps are finished, a concrete model with fixed size is established, on which model-checking is performed. If $p \in Pred$, $act \in$ `Actions`, and $\overrightarrow{\alpha}$ is a sequence of individuals of the appropriate type then $p(\overrightarrow{\alpha})$ is a ground atomic formula and $act(\overrightarrow{\alpha})$ is an instantiated action. The finite set of ground atomic formulae is P. `Actions`* is the finite set of instantiated actions. 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 as in Alloy [54] and *RW* [121].

3.2.3.1 For-loops

We describe the semantics of for-loops in the context of a model M, with $\sigma_t = \{v_1, \ldots, v_k\}$ the set of all individuals in M of the type t which the run statement defines.

Let $act \in$ `Actions`. We then transform each `for-statement` to its equivalent multiple *assignment* statements. For example the following `for-statement`:

`for` $(v : t)$ $\{p(\overrightarrow{\gamma_1}, v, \overrightarrow{\gamma_2}) := \bot;\}$

is in the write action $act(\overrightarrow{x})$ where $\overrightarrow{\gamma_1}$ and $\overrightarrow{\gamma_2}$ are subsequences of other parameters. This `for-statement` is transformed to:

$p(\overrightarrow{\gamma_1}, v_1, \overrightarrow{\gamma_2}) := \bot; \ \ldots \ p(\overrightarrow{\gamma_1}, v_k, \overrightarrow{\gamma_2}) := \bot;$

We apply this process repeatedly until we have no `for-statement` in our action. The same process will applied to all actions.

3.2.3.2 Effect of Actions

Let *act* be a loop-free action in `Actions` and $\overrightarrow{\alpha}$ a sequence of individuals of the appropriate type for *act*. We define the result of running the instantiated action $act(\overrightarrow{\alpha})$. We apply the functions: effect$^+$() and effect$^-$() which compute the positive and the negative effect of the instantiated loop-free action $act(\overrightarrow{\alpha})$ as following:

$$\text{effect}^+(act(\overrightarrow{\alpha})) = \{p(\overrightarrow{\beta}) \mid p(\overrightarrow{\beta}){:=}\top \text{ occurs in } act(\overrightarrow{\alpha})\}$$
$$\text{effect}^-(act(\overrightarrow{\alpha})) = \{p(\overrightarrow{\beta}) \mid p(\overrightarrow{\beta}){:=}\bot \text{ occurs in } act(\overrightarrow{\alpha})\}$$

where all the values of $\overrightarrow{\beta}$ are members of σ. effect$^+(act(\overrightarrow{\alpha}))$ and effect$^-(act(\overrightarrow{\alpha}))$ return two mutually exclusive sets for all $act(\overrightarrow{\alpha}) \in$ `Actions`* For example in a conference management system model, one may want to use add elements to the model using the following run statement.

run for 3 Paper, 4 Agent

The algorithm 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 ground atomic formalas: from `Author`(p_1, a_1) to `Author`(p_3, a_4). Finally, all the defined actions are instantiated the same way. For example, the action `DeletePCmember(agent : Agent)` is instantiated to the four actions:

`DeletePCmember`(a_1), `DeletePCmember`(a_2), `DeletePCmember`(a_3), `DeletePCmember`(a_4)

Subsequently, effect$^-$(`DeletePCmember`(a_1)) is the set:

$\{\texttt{PCmember}(a_1), \texttt{Chair}(a_1), \texttt{Reviewer}(p_1, a_1), \texttt{Reviewer}(p_2, a_1), \texttt{Reviewer}(p_3, a_1)\ \}$

while effect$^+$($\texttt{DeletePCmember}(a_1)$) $=\emptyset$.

3.2.4 Query

Following the run-statement, a query can be specified based on [121]. A query, taking the form of

$$\text{check}\ \{\ L\ \ ||\ I \rightarrow\ C_1 : (G_1\ \texttt{THEN}\ C_2 : (G_2\ \ \ldots\ \ \texttt{THEN}\ C_n : (G_n) \ldots))\}$$

where L, defines a number of quantified variables used in I, G_i and C_i; I (optional) is a list of conditions based on which the goal, defined by G_i, is to be achieved; and C_i defines a coalition of agents who work together, intending to achieve the goal. Its meaning is: Are there strategies available for agents in C_1, C_2, ..., and C_n, such that, if conditions in I are true, the agents in C_1 can achieve a state in which the goal G_1 is satisfied, and then (in that state) the agents in C_2 can achieve the goal G_2, ..., and finally the agents in C_n can achieve the goal G_n? What this nested goal describes is a sequencing of actions performed by the agents in C_1, C_2, ..., and C_n. The check statement goes through an instantiation process where the quantified variable defined in L and used in I, G_i and C_i are replaced with the appropriate individuals from σ to conform with model similar to the way we handle actions and atomic formulae in the previous example.

The syntax of L can be further specified as following:

$$\begin{aligned} \texttt{V} &::= \texttt{v}\ (\texttt{,v})^*\texttt{:t} \\ \texttt{L} &::= (\textbf{A} \mid \textbf{E}\ )\ [\textbf{dist}]\ \texttt{V}\ (\texttt{,V})^* \end{aligned}$$

where `v` is a variable name and `t` is a type. Remember that `t` must have been defined in the system before hand using type definition statment. For example we can define an agent `a` of the type `Agent`. The keyword **dist** specifies the way the query can be computed and how to populate the query. When **dist** is used the algorithm will make sure that

it excludes the rounds where different variables defined on the same type play the same element. We discuss this in detail with an explanation of the computational rounds in detail is Sction 6.5. `V` defines a number of agents of the type `t`. G can be:

a *reading goal* which takes the form of C :[p($\overrightarrow{y}$)] which means an agent or more in the coalition `C` know that they can *read* the variable `p(`$\overrightarrow{y}$`)` where `p(`$\overrightarrow{y}$`)`$\in P$.

a *making goal* which takes the form {`p(`$\overrightarrow{y}$`)`}which means an agent or more in the coalition `C` can reach a state where they know that they can *write* `p(`$\overrightarrow{y}$`)` to true where `p(`$\overrightarrow{y}$`)`$\in P$.. *making goal* is distinguished by the brackets {}.

C_i is the set of agents involved in the coalition while I specify the intial state. I is a logical formula constructed from the variables in P followed by either '!' which means that these variables in I are known to the agents in C_i. On the other hand variables followed by '*!' mean that the coalition know the value of that variable but must not change that value during the strategy. Note that variables used in C_i and $\overrightarrow{y}$ have to be declared in L. We revisit this notation when we discuss the representation of the initial knowledge state in the next Chapter, Section 4.2.

Example: The following one level nested query

```
check {E dist a,b,c: Agent, p: Paper ||
        Chair(c)*! & ~Author(p,a)*! and
        Submitted-review(p,b,c)*! and
        ~Submitted-review(p,a,b)! 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,c)] THEN {a,c}:({Submittedreview(p,a,b)}))}
```

represents the property that a reviewer `a` should not read the review of another reviewer before she submits her own review for that paper. The variables followed by '!' mean that these variables in I are known to the agents in C_i, in this case agents `a` and `c`. On the other hand variables followed by '*!' mean that the coalition know the value of that variable but must not change that value during the strategy. Similarly, we say that G_1=`[Review(p,b,c)]` is a *reading goal* which means agent `a` knows that she can *read* the variable `Review(p,b,c)` while G_2={`Submittedreview(p,b,c)`} means that agent `a` and `c` can reach a state where they know that they can *write* `Submittedreview(p,a,b)` to true as it is a *making goal* which we distinguish by the brackets {}.

More query examples are provided in Chapters 6 and 5.

3.2.5 Strategy

A strategy is a sequence of read or write actions where in each step, there is an agent in $\sigma_{\mathtt{C}}$ who can execute an action.

`Strategy`::= null | a:$act(\overrightarrow{\alpha})$; | `Strategy Strategy`

| if($p(\overrightarrow{\alpha})$){`Strategy`} else {`Strategy`}

where $a \in \sigma_{\mathtt{C}}$, $p \in Pred$ and $act \in$ `Actions` and α is a sequence of individuals where its members are in σ. Note that only one agent can act at a time as the agents perform actions asynchronously which is a realistic assumption in computer systems.

3.3 Discussion and Summary

In this chapter we have introduced the syntax of *X-Policy* modelling language, its syntax and informal semantics. We also provided examples and discussed the language expressiveness by expressing dynamic access control policies idioms in *X-Policy*. We also introduced the *X-Policy* query language and provided some examples.

In Section 3.2.2 a number of access control idioms are expressed in *X-Policy* like mutual exclusivity, inheritance and atomic actions. However, *X-Policy* does not express

other constrains (e.g. role cardinality constraints). For example, we can not define rules where we say for example that each paper should have at least three reviews. In addition *X-Policy* does not allow conditional bulk insertion where can specify dynamic conditions for example delete all papers where the author is a PCmember. This requires the ability to specify conditions over the for-loop. In the case where no read permissions for a variable is explicitly specified in *X-Policy*, *X-Policy* will treat it as a prohibited variable where no agent is allowed to read it. In the case of inconsistency, for example a variable read permission is defined twice, the algorithm will through an error as it only allow each action execution rule and each variables read permission rule to be specified once.

In the next chapter we will discuss the model checking algorithm and how we analyse *X-Policy* queries. In Chapter 5 we model and analyse a number of web-based conference management systems.

CHAPTER 4

X-Policy Model Checking

Chapter 3 introduced the *X-Policy* modelling language. *X-Policy* allows us to specify a query Q, such as the one defined in 3.2.4. The task of the *X-Policy* model-checking algorithm is to figure out whether or not the strategies queried by Q exist, and if they exist, output at least one of them. The algorithm performs a backward reachability search to find a strategy that can allow the coalition agents defined in the query to go from the initial state to the goal state. The algorithm uses agent's knowledge about the system state to analyse the system policy and make decisions to achieve the goal.

In this Chapter, the knowledge transition system is described in Section 4.1. In Section 4.2 the agent's knowledg representation of the initial state is illustrated while the knowledge representation of the goal state is discussed in Section 4.3. Section 4.4 describes the backwards reachability computation where the process of computing the set of states, generating strategies and the pseudo-code for the algorithm is detailed. In Sections sec:rwproof and 4.6 we argue about the algorithm correctness and the computational complexity of the algorithm. A summary of the Chapter is included in Section 4.7.

4.1 The transition system

The algorithm is built around the knowledge of the state of the system S that the considered coalition C has at each step of implementing its strategy. This is achieved by extending the idea of [121] to handle complex executable actions instead of simple assignments.

Obviously there is a set of knowledge states each of which is sufficient for C to regard its goal as achieved. This is so when C knows that the formulae in some appropriate combination of the involved making goals are true, or that enough is known to work out the truth values of the formulae in the reading goals. We denote the set of the knowledge states from which C can deduce that its goal is achieved by $K_{\mathtt{G}}$. Each step of a strategy takes C from a knowledge state to a possibly richer one until a state in $K_{\mathtt{G}}$ is reached. A knowledge state combines knowledge of the initial state of the system, that is, the state of the system at the beginning of executing a strategy, and knowledge of its current state. Executing actions contribute the knowledge of the current values of the assigned variables, which has been just given to them. This means that learning and changing the system are done simultaneously. Reading steps or as we also call it sampling steps contribute C's knowledge on both the current and the previous value of the sampled variable before the sampling. Overwriting a variable without knowing its value (i. e. reading it values in advance) destroys the prospect to learn the original values of the variables before the overwriting. Strategies are supposed to take C from its initial knowledge state $k_{\mathtt{init}}$ to one in $K_{\mathtt{G}}$ from which its goal is deemed as achieved.

To describe C's knowledge on $p(\overrightarrow{\alpha})$, we use four knowledge variables. For each $p(\overrightarrow{\alpha}) \in P$, where P is the set of finite set of atomic formulae as defined in Section 3.2.3, we then

have

$v_{0p(\overrightarrow{\alpha})}$	is true if C knows the original value of $p(\overrightarrow{\alpha})$
$t_{0p(\overrightarrow{\alpha})}$	is true if C knows the original $p(\overrightarrow{\alpha})$ is true
$v_{p(\overrightarrow{\alpha})}$	is true if C knows the current value of $p(\overrightarrow{\alpha})$
$t_{p(\overrightarrow{\alpha})}$	is true if C knows currently $p(\overrightarrow{\alpha})$ is true

When overwriting $p(\overrightarrow{\alpha})$ to true, $v_{p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ both become true, but $v_{0p(\overrightarrow{\alpha})}$ and $t_{0p(\overrightarrow{\alpha})}$ do not change, because overwriting does not contribute C's knowledge on $p(\overrightarrow{\alpha})$'s previous value. When overwriting $p(\overrightarrow{\alpha})$ to false, $v_{p(\overrightarrow{\alpha})}$ becomes true; $t_{p(\overrightarrow{\alpha})}$ becomes false; both $v_{0p(\overrightarrow{\alpha})}$ and $t_{0p(\overrightarrow{\alpha})}$ do not change. When sampling $p(\overrightarrow{\alpha})$, $v_{0p(\overrightarrow{\alpha})}$ and $v_{p(\overrightarrow{\alpha})}$ both become true and $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ both become false if $p(\overrightarrow{\alpha})$ turns out to be false, or $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ both become true if $p(\overrightarrow{\alpha})$ turns out to be true.

The reason that the values of both $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ are changed when reading the current value to indicate that no change has occurred on that value which distinguish it from the overwriting scenario. This is possible because we assume that the agents have an exclusive access to the system during our analysis so the value of these variables does not change unless one of the agents overwrites that variable. Since the contents of $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ are irrelevant when $p(\overrightarrow{\alpha})$ 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 $v_{0p(\overrightarrow{\alpha})}$, $t_{0p(\overrightarrow{\alpha})}$, $v_{p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ as independent variables. The reason that the values of both $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ are changed when reading the current value to indicate that no change has occurred on that value which distinguish it from the overwriting scenario. This is possible because we assume that the agents have an exclusive access to the system during our analysis so the value of these variables does not change unless one of the agents overwrites that variable. Since the contents of $t_{0p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ are irrelevant when $p(\overrightarrow{\alpha})$

is unknown, and the original 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 $v_{0p(\overrightarrow{\alpha})}$, $t_{0p(\overrightarrow{\alpha})}$, $v_{p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ as independent variables.

A knowledge state is given by the quadruple (V_0, T_0, V, T), where $V_0 = \{p(\overrightarrow{\alpha}) \in P \mid v_{0p(\overrightarrow{\alpha})}\ \texttt{is}\ \top\}$, $T_0 = \{p(\overrightarrow{\alpha}) \in P \mid t_{0p(\overrightarrow{\alpha})}\ \texttt{is}\ \top\}$, $V = \{p(\overrightarrow{\alpha}) \in P \mid v_{p(\overrightarrow{\alpha})}\ \texttt{is}\ \top\}$, and $T = \{p(\overrightarrow{\alpha}) \in P \mid t_{p(\overrightarrow{\alpha})}\ \texttt{is}\ \top\}$.

4.2 Representation of $k_{\texttt{init}}$

$k_{\texttt{init}}$ is the state where C knows nothing about the state of S, except on the values of the variables marked by '!' in the conditions defined by I in the Query using the syntax defined in Section 3.2.4. This means that for all members of P which also occur in I and are marked by '!', C knows their values. However, for all the other members of P, C does not know their values initially. Now for each variable $p(\overrightarrow{\alpha}) \in P$, we have four knowledge variables $v_{0p(\overrightarrow{\alpha})}$, $t_{0p(\overrightarrow{\alpha})}$, $v_{p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ to describe C's original and current knowledge about it. We use a boolean expression composed of the knowledge variables to represent $k_{\texttt{init}}$. This boolean expression is then represented by a BDD in the course of the *X-Policy* model-checking.

We divide members of P into three mutually-exclusive subsets. P^+ is the set for members of P which only occur positively in I and are marked by '!' and by '*!'. P^- is the set for members of P which only occur negatively in I and are marked by '!' and by '*!'. P° is the set for all the other members of P. Now for each $p(\overrightarrow{\alpha}) \in P^+$, we use $(v_{0p(\overrightarrow{\alpha})} \wedge t_{0p(\overrightarrow{\alpha})} \wedge v_{p(\overrightarrow{\alpha})} \wedge t_{p(\overrightarrow{\alpha})})$ to represent C's initial knowledge about $p(\overrightarrow{\alpha})$. In the case that $p(\overrightarrow{\alpha}) \in P^-$, we use $(v_{0p(\overrightarrow{\alpha})} \wedge \neg t_{0p(\overrightarrow{\alpha})} \wedge v_{p(\overrightarrow{\alpha})} \wedge \neg t_{p(\overrightarrow{\alpha})})$ to represent C's initial knowledge about it; and finally, if $p \in P^\circ$, we use $(\neg v_{0p(\overrightarrow{\alpha})} \wedge \neg v_{p(\overrightarrow{\alpha})})$ to represent C's initial knowledge about it given that the value of $t_{p(\overrightarrow{\alpha})}$) and $t_{0p(\overrightarrow{\alpha})}$ are irrelevant when $v_{p(\overrightarrow{\alpha})}$ and

$v_{0p(\overrightarrow{\alpha})}$ are false. Therefore, the representation of $k_{\mathtt{init}}$ by the knowledge variables is the conjunction of all the representations of the above forms for members of P.

4.3 Representation of $K_{\mathtt{G}}$

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

We want to represent G as a set of knowledge states, being those in which the goal is known to be true. A set of knowledge states can be represented by a formula over the propositions $\{v_{0p(\overrightarrow{\alpha})}, t_{0p(\overrightarrow{\alpha})}, v_{p(\overrightarrow{\alpha})}, t_{p(\overrightarrow{\alpha})} \mid p(\overrightarrow{\alpha}) \in P\}$. Note that $v_{0p(\overrightarrow{\alpha})}$ is true in a state if $p \in V_0$, $t_{0p(\overrightarrow{\alpha})}$ is true if $p(\overrightarrow{\alpha}) \in T_0$, and similarly for $v_p(\overrightarrow{\alpha})$ and $t_{p(\overrightarrow{\alpha})}$.

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

- if $v_{p(\overrightarrow{\alpha})}$ is true, then substitute $t_{p(\overrightarrow{\alpha})}$ for $p(\overrightarrow{\alpha})$ in l. This covers the case that the agent knows the value of $p(\overrightarrow{\alpha})$.

- if $v_{p(\overrightarrow{\alpha})}$ is false, then replace l with a version in which $\top$ is substituted for $p(\overrightarrow{\alpha})$ and another in which $\bot$ is substituted for $p(\overrightarrow{\alpha})$. This covers the case that the agent does not know $p(\overrightarrow{\alpha})$.

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

$$(l[t_{p(\overrightarrow{\alpha})}/p(\overrightarrow{\alpha})] \wedge v_{p(\overrightarrow{\alpha})}) \vee (l[\top/p(\overrightarrow{\alpha})] \wedge l[\bot/p(\overrightarrow{\alpha})] \wedge \neg v_{p(\overrightarrow{\alpha})})$$

In the following definition, we generalise this formula to consider the effect of all the $p(\overrightarrow{\alpha}) \in P$.

Definition 4.3.1 *Let V, T be the knowledge held by an agent, and l a formula over the propositions P. The propositions $v_{p(\overrightarrow{\alpha})}$ and $t_{p(\overrightarrow{\alpha})}$ signify $p(\overrightarrow{\alpha}) \in V$ and $p(\overrightarrow{\alpha}) \in T$, respectively. The $\gamma_{V,T}l$ is represented in the formula:*

$$\Big(\bigwedge_{S \subseteq P} l[(v_{p(\overrightarrow{\alpha})} \rightarrow p(\overrightarrow{\alpha}))/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in S] \\ [(v_{p(\overrightarrow{\alpha})} \wedge p(\overrightarrow{\alpha}))/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in P \setminus S] \Big) \\ [t_{p(\overrightarrow{\alpha})}/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in P]$$

$\gamma_{V,T}l$ expresses the agent's ability to determine that l is true. S is the set of all possible subsets of P. Where $\bigwedge_{S \subseteq P}$ is a loop over all the possible values of $S \subseteq P$. The $S \subseteq P$ produces all the possible combinations of $\top$ and $\bot$ to substitute $p(\overrightarrow{\alpha})$s such that $v_{p(\overrightarrow{\alpha})}$ is false. Note that $v_{p(\overrightarrow{\alpha})} \rightarrow p(\overrightarrow{\alpha})$ is $\top$ when $v_{p(\overrightarrow{\alpha})}$ is false, and $p(\overrightarrow{\alpha})$ otherwise; similarly, $v_{p(\overrightarrow{\alpha})} \wedge p(\overrightarrow{\alpha})$ is $\bot$ if $v_{p(\overrightarrow{\alpha})}$ is false and $p(\overrightarrow{\alpha})$ otherwise. Hence, S just enumerates all the vectors of $\top$s and $\bot$s for the $p(\overrightarrow{\alpha})$s and $v_{p(\overrightarrow{\alpha})} \rightarrow p(\overrightarrow{\alpha})$ and $v_{p(\overrightarrow{\alpha})} \wedge p(\overrightarrow{\alpha})$ are used to restrict the effect of the substitution only to $p(\overrightarrow{\alpha})$s such that $v_{p(\overrightarrow{\alpha})}$ is false.

Example 4.3.1 *Let $P = \{p, q\}$, $l = q \wedge \neg p$ and $S \subseteq P$ and assuming that we have only one coalition of one agent, then $P \setminus S \in \{\{p, q\}, \{p\}, \{q\}, \{\}\}$. The $\gamma_{V,T}l$ in this case can be produced first by computing the terms of the formula over possible values of S:*

$$l[(v_{p(\overrightarrow{\alpha})} \rightarrow p(\overrightarrow{\alpha}))/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in S][(v_{p(\overrightarrow{\alpha})} \wedge p(\overrightarrow{\alpha}))/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in P \setminus S]$$

as following:

$P = \{p, q\}$	$S = \{\}$	$(v_q \wedge q \wedge \neg(v_p \wedge p))$
$P = \{q\}$	$S = \{p\}$	$(v_q \wedge q \wedge \neg(v_p \rightarrow p))$
$P = \{p\}$	$S = \{q\}$	$(v_q \rightarrow q \wedge \neg(v_p \wedge p))$
$P = \{\}$	$S = \{p, q\}$	$(v_q \rightarrow q \wedge \neg(v_p \rightarrow p))$

Now we apply the $\bigwedge_{S \subseteq P}$ *operation which will produce the following formula after the first level of substitution:*

$$\Big(v_q \wedge q \wedge \neg(v_p \wedge p) \wedge v_q \rightarrow q \wedge \neg(v_p \wedge p) \wedge v_q \wedge q \wedge \neg(v_p \rightarrow p) \wedge v_q \rightarrow q \wedge \neg(v_p \rightarrow p)\Big)$$
$$[t_{p(\overrightarrow{\alpha})}/p(\overrightarrow{\alpha}) \mid p(\overrightarrow{\alpha}) \in P]$$

We apply the substitution of $p(\overrightarrow{\alpha})$ *with* $t_{p(\overrightarrow{\alpha})}$ *for all* $p(\overrightarrow{\alpha}) \in P$ *which produces:*

$$\Big(v_q \wedge t_q \wedge \neg(v_p \wedge t_p) \wedge v_q \rightarrow t_q \wedge \neg(v_p \wedge t_p) \wedge v_q \wedge t_q \wedge \neg(v_p \rightarrow t_p) \wedge v_q \rightarrow t_q \wedge \neg(v_p \rightarrow t_p)\Big)$$

This can be then simplified to using Boolean identity:

$$v_q \wedge t_q \wedge \neg(v_p \wedge t_p) \wedge v_q \rightarrow t_q \wedge \neg(v_p \rightarrow t_p) \wedge \neg(v_p \rightarrow t_p)$$

and finally it can be written as:

$$v_q \wedge t_q \wedge v_p \wedge \neg t_p$$

Where this formula evaluates whether the agent knows if l *is true.*

4.3.1 Substitution of reading goals

The knowledge states in which the reading goal $[l]$ is known to be achieved are those in which the knowledge held is sufficient to evaluate l in V_0, T_0. In order to do that, the agent

needs to be able to determine that l is true, or that it is false. Thus, the appropriate formula over $\{v_{0p(\overrightarrow{\alpha})}, t_{0p(\overrightarrow{\alpha})}, v_{p(\overrightarrow{\alpha})}, t_{p(\overrightarrow{\alpha})} \mid p(\overrightarrow{\alpha}) \in P\}$ is $\gamma_{V_0,T_0} l \vee \gamma_{V_0,T_0}(\neg l)$.

Example 4.3.2 *Similar to Example 4.3.1, if* $P = \{p, q\}$, $l = q \wedge \neg p$ *and* $S \subseteq P$ *and assuming that we have only one coalition of one agent, then* $P \setminus S \in \{\{p, q\}, \{p\}, \{q\}, \{\}\}$. *The* $\gamma_{V_0,T_0} l \vee \gamma_{V_0,T_0}(\neg l)$ *in this case will produce the following formula:*

$$(v_{0q} \wedge \neg t_{0q}) \vee (v_{0p} \wedge t_{0p}) \vee (v_{0q} \wedge t_{0p} \wedge v_{0p})$$

which covers all the possible knowledge states in which the agents can evaluate the value of the goal $[l]$*:*

- *The agent knows that the value of the variable* q *is* $\bot$ *and therefore she can deduce that the value of* $[l]$ *which is* $\bot$.
- *The agent knows that the value of the variable* p *is* $\bot$ *and therefore she can deduce that the value of* $[l]$ *which is* $\bot$.
- *The agent knows that the value of the variable* q *is* $\top$ *and value of the variable* p *and whether it is* $\top/\bot$ *and therefore she can deduce that the value of* $[l]$ *which is* $\top/\bot$ *respectively.*

4.3.2 Substitution of making goals

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

Example 4.3.3 *From Example 4.3.1 The* $\gamma_{V,T} l$ *in this case is written as:*

$$v_q \wedge t_q \wedge v_p \wedge \neg t_p$$

which represents the knowledge state in which the agent has reached a state where the value of the system variables satisfy the goal l.

The formula $\gamma_{V,T}l$ is first defined by [42] and since has been used by *RW* [121], a precursor of *X-Policy*. *X-Policy* uses the same definition. In this thesis we contribute an illustration of that definition in the Examples 4.3.1, 4.3.2 and 4.3.3.

4.4 Backwards reachability computation

4.4.1 Computing sets of states

This is a crucial step that shows how we deal with compound actions in our reachability analysis. To find strategies the algorithm starts from K_G, searching for sets of the knowledge states which transition into K_G by reading a variable $p(\overrightarrow{\alpha}) \in P$; sets of the knowledge states which transition into $K_{\mathtt{G}}$ by executing a write action $act(\overrightarrow{\alpha}) \in \mathtt{Actions}^*$; 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 two kinds of transition relations. During this process, if $k_{\mathtt{init}}$ is found in a set of knowledge states, the goal is considered as reachable by following the operations represented by the transition relations which connect the set in which $k_{\mathtt{init}}$ is found to $K_{\mathtt{G}}$. The operations along the path are deemed as the steps of a strategy by the algorithm.

We also use $\gamma_{V,T}\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a) = \top$ and $\gamma_{V,T}\mathtt{r}(\mathtt{p}(\overrightarrow{\alpha}), a) = \top$ to denote the conditions under which a knows with Knowledge V, T that she is permitted to execute $\mathtt{act}(\overrightarrow{\alpha})$ and read $p(\overrightarrow{\alpha})$ respectively. The mapping $\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a)$ and $\mathtt{r}(p(\overrightarrow{\alpha}), a)$ are boolean expressions composed of members of P as defined in the system policy. $\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a)$ defines the condition under which a is permitted to execute $\mathtt{act}(\overrightarrow{\alpha})$. $\mathtt{r}(p(\overrightarrow{\alpha}), a)$ defines the condition under which a is permitted to read $p(\overrightarrow{\alpha})$. For instance, to represent a's current knowledge on $\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a)$, we need to use the knowledge variables in V and T to replace

every occurrence of variables in $\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a)$, because variables in V and T describes C's knowledge (a and C share the same knowledge) about the current state of the system. The same applies to $\mathtt{r}(p(\overrightarrow{\alpha}), a)$.

In order to formally describe the process of solving the reachability problem, we shall first define the concept of pre-sets. For any $a \in C$, $p(\overrightarrow{\alpha}) \in P$ and and $act(\overrightarrow{\alpha}) \in \mathtt{Actions}^*$, where a given set of knowledge states Y.

- $\mathtt{Pre}^{\exists,a}_{\mathtt{act}(\overrightarrow{\alpha})}(Y)$ is the set of the knowledge states in which a knows she is permitted to execute $\mathtt{act}(\overrightarrow{\alpha})$ and which transition into Y by executing the action $\mathtt{act}(\overrightarrow{\alpha})$. Its formal definition is:

  $\{(V_0, T_0, V, T) \mid (V_0, T_0, V \cup \text{effect}^+(act(\overrightarrow{\alpha})) \cup \text{effect}^-(act(\overrightarrow{\alpha})), T \cup \text{effect}^+(act(\overrightarrow{\alpha})) \setminus \text{effect}^-(act(\overrightarrow{\alpha}))) \in Y, \gamma_{V,T}\mathtt{X}(\mathtt{act}(\overrightarrow{\alpha}), a) = \top\}$.

  Note that in a write action Y differs from the previous state as following:

  V_0 does not change as the agent knowledge about the original value of the variables remains the same during a write action.

  T_0 does not change as the original values of the variables does not change during a write action.

  V includes all the variables mentioned in the action as the agent will know their value given that the agent has an exclusive access to the system and is the one executing the action.

  T includes all the variables which are updated to $\top$ in the action and will omit all the variables that are updated to $\bot$.

- $\mathtt{Pre}^{\exists,a}_{\mathtt{p}(\overrightarrow{\alpha})=\top}(Y)$ is the set of the knowledge states in which a knows she is permitted to read the value of the variable $\mathtt{p}(\overrightarrow{\alpha})$ and which transition into Y by returning the value of $\mathtt{p}(\overrightarrow{\alpha})$ and find out it is true ($\top$). Its formal definition is:

$\{(V_0, T_0, V, T) \mid \mathtt{p}(\overrightarrow{\alpha}) \notin V_0, \mathtt{p}(\overrightarrow{\alpha}) \notin V, (V_0 \cup \{\mathtt{p}(\overrightarrow{\alpha})\}, T_0 \cup \{\mathtt{p}(\overrightarrow{\alpha})\}, V \cup \{\mathtt{p}(\overrightarrow{\alpha})\}, T \cup \{\mathtt{p}(\overrightarrow{\alpha})\}) \in Y, \gamma_{V,T}\mathtt{r}(\mathtt{p}(\overrightarrow{\alpha}), a) = \top\}$.

Note that when an agent reads a variable $\mathtt{p}(\overrightarrow{\alpha})$, Y differs from the previous state as following:

V_0 and V include $\mathtt{p}(\overrightarrow{\alpha})$ as the agent now knows its value and given that a read operation does not change the value of the variable.

T_0 and T include $\mathtt{p}(\overrightarrow{\alpha})$ as the agent finds out that its value is $\top$

- $\mathtt{Pre}^{\exists,a}_{\mathtt{p}(\overrightarrow{\alpha})=\bot}(Y)$ is the set of the knowledge states in which a knows she is permitted to read the value of the variable $\mathtt{p}(\overrightarrow{\alpha})$ and which transition into Y by returning the value of $\mathtt{p}(\overrightarrow{\alpha})$ and find out it is false ($\bot$). Its formal definition is:

  $\{(V_0, T_0, V, T) \mid \mathtt{p}(\overrightarrow{\alpha}) \notin V_0, \mathtt{p}(\overrightarrow{\alpha}) \notin V, (V_0 \cup \{\mathtt{p}(\overrightarrow{\alpha})\}, T_0 \setminus \{\mathtt{p}(\overrightarrow{\alpha})\}, V \cup \{\mathtt{p}(\overrightarrow{\alpha})\}, T \setminus \{\mathtt{p}(\overrightarrow{\alpha})\}) \in Y, \gamma_{V,T}\mathtt{r}(\mathtt{p}(\overrightarrow{\alpha}), a) = \top\}$.

  Note that when an agent reads a variable $\mathtt{p}(\overrightarrow{\alpha})$, Y differs from the previous state as following:

  V_0 and V include $\mathtt{p}(\overrightarrow{\alpha})$ as the agent now knows its value and given that a read operation does not change the value of the variable.

  T_0 and T omit $\mathtt{p}(\overrightarrow{\alpha})$ as the agent finds out that its value is $\bot$

4.4.2 Generating strategies

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

The algorithm starts with the pair $(K_{\mathtt{G}}, \mathtt{skip;})$. The core of the algorithm works as follows: Given the pair (Y, s), it adds the pairs $(\mathtt{Pre}^{\exists,a}_{\mathtt{act}(\overrightarrow{\alpha})}(Y), (\mathtt{a{:}act}(\overrightarrow{\alpha}); s))$. For any

two pairs (Y_1, s_1) and (Y_2, s_2) where $\texttt{Pre}^{\exists,a}_{\texttt{p}(\overrightarrow{\alpha})=\top}(Y_1) \cap \texttt{Pre}^{\exists,a}_{\texttt{p}(\overrightarrow{\alpha})=\bot}(Y_2) = Y$, it adds the pair $(\texttt{Pre}^{\exists,a}_{\texttt{p}(\overrightarrow{\alpha})=\top}(Y_1) \cap \texttt{Pre}^{\exists,a}_{\texttt{p}(\overrightarrow{\alpha})=\bot}(Y_2), \textbf{if } (\texttt{a} : \texttt{p}(\overrightarrow{\alpha})) \textbf{ then } s_1 \textbf{ else } s_2)$. These two steps are repeated repeatedly for all $\texttt{p}(\overrightarrow{\alpha})$ in P, $act(\overrightarrow{\alpha}) \in \texttt{Actions}^*$, and $a \in C$ until the set of `strategies` does not change any more.

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

4.4.3 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 $k_{\texttt{init}}$ and the set of goal knowledge states $K_{\texttt{G}}$. It outputs at least a strategy if there is one which can derive the model going from $k_{\texttt{init}}$ to some state in $K_{\texttt{G}}$. The algorithm works by backwards reachability from $K_{\texttt{G}}$ to $k_{\texttt{init}}$. It maintains a set of states it has seen, called `states_seen`, and a data structure storing the pairs found so far, called `strategies`.

```
1  strategies := ∅;
2  states_seen := ∅;
3  put (K_G, skip;) in strategies;
4  repeat until strategies does not change{
5          choose (Y_1, s_1) ∈ strategies;
6          for each act(α⃗) ∈ Actions* {
7                  for each a ∈ C{
8                  PXY := Pre^{∃,a}_{act(α⃗)}(Y_1);
9                  if ((PXY ≠ ∅) ∧ (PXY ⊈ states_seen)){
10                         states_seen := states_seen ∪ PXY;
11                         axs_1 := "a : act(α⃗) "+ s_1;
12                         strategies := strategies ∪ {(PXY, axs_1)};
13                         if (k_init ∈ PXY)
14                                 output axs_1;
15                         }
```

```
16                  }
17          }
18          choose (Y_2, s_2) ∈ strategies;
19          for each variable p(α⃗) ∈ P{
20                  for each a ∈ C{
21                  PRY := Pre^{∃,a}_{p(α⃗)=⊤}(Y_1) ∩ Pre^{∃,a}_{p(α⃗)=⊥}(Y_2);
22                  if ((PRY ≠ ∅) ∧ (PRY ⊈ states_seen)){
23                          states_seen := states_seen ∪ PRY;
24                          pss := "if (a : p(α⃗)) then s_1 else s_2";
25                          strategies := strategies ∪ {(PRY, pss)};
26                          if (k_init ∈ PRY)
27                                  output pss;
28                          }
29                  }
30          }
31  }
```

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

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

4.5 Argument about correctness

Claim 4.5.1 *The algorithm described in Section 4.4.3 will eventually terminate.*

Argument. 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 in the model, we cannot go on encountering new states for ever. Note that we assume tat the size of `strategies` is not shrinking as the algorithm only add to the `strategies`. □

Claim 4.5.2 *If there exists a strategy s that drives the system from k_{init} to K_G, then the algorithm will produce at least one strategy (but not necessarily s).*

Argument. The algorithm starts from $K_{\texttt{G}}$ and consider all the states which can lead to $K_{\texttt{G}}$ by executing an action by an agent. This is done recursively until it covers all possible actions and all possible agents. This operation is repeated over all the discovered states until no new states are discovered. Which by definition means that the algorithm performs an exhaustive backwards reachability and therefore will find all states for which there is a strategy to arrive at $K_{\texttt{G}}$.

Since s is a strategy from $k_{\texttt{init}}$ to $K_{\texttt{G}}$, $k_{\texttt{init}}$ will be in the set of states found and therefore the algorithm will output a strategy from $k_{\texttt{init}}$ to $K_{\texttt{G}}$. As however the choice operator is resolved, $k_{\texttt{init}}$ will eventually be included in `states_seen`, and therefore some strategy will be generated. □

Remark. The algorithm is non-deterministic thanks to the choice operator. We mean by non-deterministic that it is not guaranteed to obtain the same s each time the algorithm is run (as we state in the Claim 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. □

Sub-claim 4.5.1 *For all $(Y, s) \in$ `strategies`, and for all $y \in Y$, s succeeds on y and the result is in $K_{\mathtt{G}}$.*

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

(i) $(\mathrm{PXY}, axs_1))$ is added, where, $\exists\ a \in A$ and $act(\overrightarrow{\alpha}) \in \mathtt{Actions}^*$, such that $PXY = \mathtt{Pre}^{\exists,a}_{\mathtt{act}(\overrightarrow{\alpha})}(Y)$, $axs_1 =$ "$\mathtt{a} : \mathtt{act}(\overrightarrow{\alpha})$ "$+\ s_1$, and (Y_1, s_1) is in `strategies`.

The inductive hypothesis states that for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in $K_{\mathtt{G}}$. We know for all $y \in PXY$ that a can execute $\mathtt{act}(\overrightarrow{\alpha})$ and that the result of that is in Y_1, because that is the way we get PTY_1 from Y_1. Therefore axs_1 succeeds on all the states in PTY_1 and the result is in $K_{\mathtt{G}}$.

(ii) (PRY, pss) is added, where, $\exists\ a \in A$ and $p(\overrightarrow{\alpha}) \in P$, such that $PSY = \mathtt{Pre}^{\exists,a}_{(\overrightarrow{\alpha})=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{(\overrightarrow{\alpha})=\bot}(Y_2)$, $pss =$ "if $(a : p(\overrightarrow{\alpha}))$ then s_1 else s_2" and (Y_1, s_1), and (Y_2, s_2) are both in `strategies`.

The inductive hypothesis states that for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in $K_{\mathtt{G}}$, and $y_2 \in Y_2$, s_2 succeeds on y_2 and result is in $K_{\mathtt{G}}$. We also know for all $y \in PRY$ that a can read $p(\overrightarrow{\alpha})$ 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 *pss* succeeds on all the states in PRY and the result is in $K_{\mathtt{G}}$. □

Claim 4.5.3 *If the algorithm outputs the strategy s then s succeeds on k_{init} and the result is in K_G.*

Argument. From Sub-claim 4.5.1 we know that for all $(Y, s) \in \mathtt{strategies}$ and $y \in Y$, s succeeds on y and the result is in $K_{\mathtt{G}}$. Because if s gets output, there must exist a Y, such that $k_{\mathtt{init}} \in Y$ and $(Y, s) \in \mathtt{strategies}$. Therefore, it follows that s succeeds on $k_{\mathtt{init}}$ and the result is in $K_{\mathtt{G}}$. □

From the implication of Claim 4.5.3, we know that if there is no strategy s which succeeds on $k_{\mathtt{init}}$ and results in $K_{\mathtt{G}}$, the algorithm will output none.

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

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

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

- Suppose $B_{\rightarrow}$ is a BDD representing one of the transition relations presented in 4.4.1, the number of nodes in of the combined $B_{\rightarrow}$ is at most $16|P|$. $B_{\rightarrow}$ is obtained by synthesising all the conditions in the formal definition $\texttt{Pre}^{\exists,a}_{\rightarrow}$.

- 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. This is possible because every boolean function has a unique, canonical BDD representation. As sharing of BDD nodes is enforced in BuDDy, equality of two functions can be decided in constant time by checking for pointer equality [80].

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

Line 5 Because the conditions in Lines 9 and 22 prevent any subset whose elements are already found from being added to `strategies`, and, in the worst case, the subsets of K are just singletons, there are at most $|K|$ $(2^{4|P|})$ pairs in `strategies`. Therefore, this step repeats at most $2^{4|P|}$ times.

Line 6 This step repeats $|\texttt{Actions}^*|$ times.

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

Line 8 According to the formula for computing $\texttt{Pre}^{\exists}_{\texttt{act}(\vec{\alpha})}(Y)$ in 4.4.1, the operation of this step involves the BDD computation $\texttt{apply}(\texttt{and},\ B_{\texttt{act}(\vec{\alpha})},\ B_{Y'_1})$ followed by an existential quantification over all the V', T' variables on the resulting BDD of the `apply` operation. The complexity of the `apply` operation in the worst case is

$16|P|2^{4|P|}$ and the complexity of the existential quantification is $2^{2|P|}$. The worse one of the two is $16|P|2^{4|P|}$. Therefore, the time spent on this step in the worst case is determined by $16|P|2^{4|P|}$.

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

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

Line 18 This step repeats at most $2^{4|P|}$ times.

Line 19 This step repeats $|P|$ times.

Line 20 This step repeats $|A|$ times.

Line 21 The time spent on this step in the worst case is determined by $16|P|2^{4|P|+1}+2^{8|P|}$. Counting the larger one of the two, the time is determined by $2^{8|P|}$.

Line 22 The time spent on this step in the worst case is determined by $2^{8|P|}$.

Line 23 The time spent on this step in the worst case is determined by $2^{8|P|}$.

Adding the time spent on each step together, we get $2^{4|P|} \times (|\texttt{Actions}^*| \times |A| \times (16|P|2^{4|P|}+2^{8|P|}+2^{8|P|})+2^{4|P|}\times|P|\times|A|\times(2^{8|P|}+2^{8|P|}+2^{8|P|}))$. Therefore, the complexity of the algorithm in the worst case is asymptotically bounded above by $3 \times 2^{16|P|}$. This is better than the time complexity of *RW* as the *X-Policy* algorithm. The compound actions in *X-Policy* allow the algorithm to direct it search better rather than exploring the possible values of the propositions. The is also confirmed by the experimental data in Table 6.1 show. It also shows the situation in practice is far better than this.

In X-*Policy* as in *RW* , there are 7 knowledge states per proposition and therefore, an access control system with n propositions contains 7^n different states. Which means that verification time and state space grow exponentially when more objects are added.

4.7 Chapter Summary

Chapter 4 discussed the model checking problem. Section 4.1 introduced the transitions system and how the knowledge variables which model the agent's knowledge of the system states. The representation of the initial knowledge and final knowledge, $k_{\mathtt{init}}$ and $K_{\mathtt{G}}$, are discussed in Section 4.2 and 4.3 including the various goal types. Section 4.4 illustrates the backward reachability computation and algorithm. Section 4.5 provides the proof of correctness. Section 4.6 analyses the computational complexity of X-*Policy* algorithm.

CHAPTER 5

Modelling Conference Management System

In this Chapter we discuss the modelling and analysing of conference management systems. We focus on the modelling of EasyChair as our case study. Our choice is motivated by the fact that EasyChair [116] is one of the largest and most used cloud based conference management system.

5.1 Overview and notes on modelling EasyChair CMS

EasyChair is a highly configurable web-based centralised conference management system that host multiple conferences. According to its website, EasyChair hosted over 4500 conferences in 2011 [116].

Conference creation and configurations. The conference organiser can request that the EasyChair administrator create a conference sub-system. When a conference creation request is made, the EasyChair administrator will manually verify the request and create a sub-system for that conference with a designated chair. EasyChair provides the conference chair with an administration page at which she may specify configuration settings for her conference. Each configuration setting affects the system access control permissions in

real time; for example the chair can open or close the submission to the conference which determines whether users can submit a paper to the system or not. The chair also can add/delete users to/from the system. The conference access control permissions also depend on the system state; for example the system prevents a paper from being assigned for reviewing to a PC member who has conflict of interest.

EasyChair account and user roles. Anyone can obtain an EasyChair account, username and password, by registering her email address which will be used globally as her identifier. Once logged in to the EasyChair system, the user will have the choice of acting as any of the roles she has been assigned to. A chair has to assign a user to be a chair or a PC member while any user can participate in the conference as an author as long as the conference submission is open. A user can only be a chair or PC member in any given conference but not both. Furthermore, within the conference system, there are three user roles: chair, PC-member or author. The EasyChair supports and manages the following processes that the conference system can go through:

- **Paper submission:** the system, in this process, collects submissions from authors. Anyone who has an account on EasyChair can submit to the conference. The submitter of the paper must be one of the paper authors. A paper can have more than one author. An author can add or delete one or more of her co-authors but not herself, whether the submission system is open or closed. The submissions can have two modes: abstract only or abstract and full text depending on whether the submission attachment is mandatory or not. The chair can view the list of the submissions and PC members if the settings allow it. At any point, the chair and/or the PC members can add any conflict of interest they might have with a certain paper. Only the chair can remove an already declared conflict of interest.

- **Paper bidding and assignment:** PC members, in this process, register their pref-

erence on which paper they like to review, often using the title and the abstract to guide their decision. Once the bidding process is done, the chair can start assigning papers for review. Once a paper is assigned to a PC member, the PC member if the reviewing menu is enabled, can view the paper. She will also receive an email message to notify her with the assignment.

- **Reviewing:** A PC member, when the system is open for reviewing, can submit her review of a paper assigned to her. PC members or the chair can request a review from someone else to subreview the paper. The reviewer cannot see other reviews on a paper assigned to her unless she submits her own review. Once she submits her review she can view other reviewers' reviews. She might also amend her review. The system will email the paper reviewers when there is a change to their review or when another review on their paper has been submitted. When all the reviews are submitted, the PC chair will ask the reviewers to start the discussion phase. In this phase, reviewers will be commenting on each other's review to try to come to a conclusion on whether to accept the paper or reject it. Only the PC chair can delete a review or a comment. The PC chair checks the conclusion of the reviewing process and then submits the decision to the system. At some point, the PC chair might decide to inform the authors of the results and email them the reviewers' feedback. The chair can always change the decision, maybe as a result of a rebuttal discussion.

The PC chair decides when to start or stop each process. One or more processes can happen at the same time. However, it is usually done in the order presented above.

5.2 Modelling conventions for EasyChair system

In this section, we discuss a number of modelling conventions we have followed in constructing ***EC***. The model ***EC*** is based on our understanding of a fragment of EasyChair.

We restrict ***EC*** to a single conference system. These conventions can be adopted to model other web-based systems.

5.2.1 System policy as set of read and write actions.

Using *X-Policy*, we specify ***EC*** as *X-Policy* actions which can be either *write actions* that change the state of the system or *read actions* that give the user/agent knowledge about the state of the system. A read action is allowed to retrieve the value of a single model variable. Actions cannot read and change the state of the system at the same time. This appears to be the way EasyChair is built. Actions are either to query the system state, or to change it, but not both. Users are only enabled to perform one operation per time. Such an assumption results in making the actions's effects independent from the state of the system and has the same effect all the time.

In EasyChair, there are some cases in which it appears that EasyChair performs a mix of read and write operations are executed in a single request. For example is when a PC member requests an agent to subreview a paper and before the agent accepts or rejects the subreviewing request, the agent can read the submission. While this seems as a combined action, it is actually two separate steps: one where the agent is read the paper and another where the agent decide to accept or reject the subreviewing assignment.

In this Chapter we discus the relevant rules in details and we include the full model in [90]. The fact that the agent has read the submission is recorded. This case is modelled in ***EC*** as two separate actions, reading the submission and recording the read. We link the two actions by allowing the sub-reviewer to read the submission if she recorded the read in advance. This is a sensible heuristic for modelling web-based systems. We also restrict our model to the system states and do not consider any possible logging system as part of our model.

5.2.2 System read operations that return multiple system values.

In systems like EasyChair, there are often forms that return multiple variables, i.e. the administrator page that return all the access options for the reviewing process etc. This page performs a number of read operations at once. We model each of these operations W as a set of actions $W_1, \ldots, W_n$. Each action W_i returns one of the values returned by W. The execution rights of W_i are the same as W. For example: EasyChair operation ShowReviews(p) which will return all the reviews on the paper p is modelled as the set of the read actions ShowReview(p,a_1)... ShowReview(p,a_n) which returns the review of agent $a_1 \ldots a_n$ on paper p.

Modelling web forms. We choose to model the fields that affect the access control policy only. For example, we do not model the operation that updates a PC member affiliation. If a page allows the user to set several fields that are mutually independent we model it as several actions. For example, whether or not the submission is closed or open and whether or not the list of submissions can be viewed by PC chairs only, PC chairs and PC members or nobody. While these two fields are related to the same subject, their values are mutually independent. There are some cases where a field can take more than two values like who is allowed to view the list of submission. In this case it can have one of the three values: PC chairs only, PC chairs and PC members, or nobody. We model such a field in our system as two variables: one to represent if PC chairs are allowed and the other to represent if PC members are allowed to view the list. To maintain the integrity of the model, we update these two variables together in an atomic action so their values are consistent with the real system behaviour. We also use atomic actions to maintain roles exclusivity, where a user for example can either be a PC chair or a PC member but not both.

Modelling paper submission and authorship management operations. We model these operations as a number of actions: SubmitPaper(Paper p,Agent a), AddAuthor(Paper p,Agent a), DeleteAuthor(Paper p,Agent a) and WithdrowSubmission(Paper p). SubmitPaper(Paper p,Agent a) models the operation of submitting the paper p for the author a by setting Author(p,a) and Submitted-paper(p,a) to true. The submitter of the paper p may or may not be the author herself, in this case agent a. We do not distinguish between the author and the author-submitter as the system treats all the authors equally once the authorship relation is established. AddAuthor(Paper p,Agent a) and DeleteAuthor(Paper p,Agent a) will establish/destroy the authorship relation between a paper p and an author a. DeleteAuthor(Paper p,Agent a) can be executed as long as the paper p has more than one author. WithdrowSubmission(Paper p) will destroy the authorship association between p and all the agents. A chair can delete a paper submission which removes it from the list of submission and list of reviews. She can also un-delete the submission which will restore it with its reviews to the lists.

5.2.3 Modelling EasyChair "log in as another pc member" functionality.

In EasyChair, the system allows the PC chair to act on behalf of another PC member using "log in as another pc member". For example a PC chair can submit a review for a paper assigned for another PC member to review. The actions executed on the PC member's behalf are indistinguishable from the ones that are executed by the PC member herself. Nevertheless, EasyChair restricts the PC chair from changing the PC member account details or accessing/editing her sub-reviewing requests. We model these actions in ***EC*** by conjoining the conditions agent user has to satisfy to act as another PC member - in this case being a chair - and the conditions that the PC member has to satisfy to execute that action. One might also consider using relations like "CanActAs", as in [10, 45]. However,

when we say A CanActAs B then we mean that A is capable of performing all the actions that B can perform which is not applicable in this case.

5.2.4 Intermediate condition.

In some cases, the system checks some intermediate conditions during an update operation like validation conditions or maintenance conditions to preserve an integrity constraint. For example, EasyChair insures that a conflict of interest is respected when a chair assigns reviewers to a paper. We express these intermediate conditions as execution preconditions. Where the checking operations reveal a system value by an error message, this value is readable by the agent requesting the operation.

5.2.5 Conference configuration settings.

We model the conference configuration settings as 0-ary atomic formulae. The value of these settings affects the conference permissions globally. In specifying the system execution policy if the user can learn about the system configuration settings from the behaviour of the system even though she cannot read the settings directly, we consider her to be able to read that variable. For example an author can learn the value of submission configuration settings by checking if she is presented with a link to submit a new paper. She is able to infer that using her knowledge of the system policy. In this case the author knows the value of the setting variable that controls whether the submission system is open or closed which is only accessible directly by the administrator. The agents use their knowledge of the system policy combined with observations of the system behaviour to evaluate the value of these variables. In some cases the user might learn partial information about a single configuration variable. For example when the list of submissions can only be viewed by PC chairs only or nobody, the PC member learns that she is not allowed to see the list of the submissions but she cannot distinguish between the two possibilities. We model this case in ***EC*** by designating a variable that represents

the fact that the PC member can read the list of submissions. The PC-member can infer the value of that variable by using the system.

5.3 *EC* model in *X-Policy* formalism

In this Section we express the ***EC*** model in *X-Policy* formalism. We first define $T = \{\mathsf{Agent}, \mathsf{Paper}\}$. To model relations between these two types, we need a number of predicates, P, as follows:

For a,b of type Agent, p of type Paper, P includes:

Chair-review-en()	Review menu is enabled for Chair. It enables Chair(s) to manage the reviews of submitted papers.
Chair-status-en()	Status menu is enabled for Chair. It enables Chair(s) to manage the status of submitted papers.
PCM-access-reviews-en()	PC members can access (view) other papers reviews.
PCM-review-editing-en()	PC members can add/modify reviews.
PCM-review-menu-en()	Review menu is enabled for PC members. It enables PC members to manage paper reviews.
PCM-status-en()	Status menu is enabled for PC members. It enables PC members to manage paper status.
Review-assig-enabled()	Review assignments enabled.
Show-reviewer-name()	Reviewer's name is readable by other PC members.
Sub-anonymous()	Submissions are anonymous. The name of authors are obscured.
Sub-open()	Submission system is open and accepts new papers.
View-sub-by-chair-permitted()	PC chairs can view the list of submissions.
View-sub-by-PCM-permitted()	PC members can view the list of submissions.
View-sub-title-permitted()	PC members can view the submission title for papers not assigned to them.
View-sub-txt-permitted()	PC members can view the submission text for papers not assigned to them.
Author(p,a)	Agent a is an author of paper p.
Chair(a)	Agent a is the chair of the PC.
Conf-of-interest(p,a)	Agent a has a conflict of interest with the paper p.
Decided-subrev(p,a,b)	Agent b has decided whether to accept or reject the subreviewing request for paper p issued by agent a.
PCmember(a)	Agent a is a PC member.
Requested-subrev(p,a,b)	Agent a has requested agent b to be his subreviewer for paper p.
Reviewer(p,a)	Paper p is assigned to PC member a for reviewing.
Submitted-paper(p,a)	Paper p is submitted by agent a.

Submitted-review(p,a,b)	Agent b's review of paper p has been submitted by agent a.
Subreviewer(p,a,b)	Agent b has accepted the subreviewing request for paper p issued by agent a.
Updated-review(p,a,b)	Agent b's review of Paper p has been updated by PC member a.

We now define the set of actions Actions and their execution permissions using the formula exec(act, user) for each action act $\in$ Actions. The execution permission statements define whether or not user of type Agent is allowed to execute such an action and in what state. We also define the read permissions rules using the formula r(prop,user) for each prop $\in P$. In the following, we list a sub-set of ***EC*** actions and their permission statements which are used in our properties analysis in *X-Policy*:

1- When the review menu is enabled and the submitted paper is not deleted: **(a)** A PC chair can read all the paper reviews. **(b)** A PC member can read a review for a paper p if she is a reviewer of that paper and has submitted her review. **(c)** A PC member can read a review for a paper to which she is not assigned, when PC members are permitted to access the titles and reviews of submitted papers. She also must have no conflict of interest with that paper.

$$
\begin{aligned}
\mathrm{r}(\text{Submitted-review}(\mathsf{p},\mathsf{a},\mathsf{b}),\text{user}) \rightleftharpoons \Bigg(& \begin{pmatrix} \text{Chair(user)} \wedge \text{Chair-review-en()} \\ \wedge \exists \mathsf{d} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{d}) \end{pmatrix} \\
\vee & \begin{pmatrix} \text{PCmember(user)} \wedge \text{Reviewer}(\mathsf{p},\text{user}) \\ \wedge \text{PCM-review-menu-en()} \\ \wedge \exists \ \mathsf{c} : \text{Agent} \ . \ \text{Submitted-review}(\mathsf{p},\text{user},\mathsf{c}) \\ \wedge \exists \mathsf{d} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{d}) \\ \wedge \neg \text{Conf-of-interest}(\mathsf{p},\text{user}) \end{pmatrix} \\
\vee & \Bigg(\begin{pmatrix} \text{PCmember(user)} \wedge \neg \text{Reviewer}(\mathsf{p},\text{user}) \\ \wedge \text{PCM-review-menu-en()} \\ \wedge \text{View-sub-title-permitted()} \\ \wedge \text{PCM-access-reviews-en()} \\ \wedge \neg \text{Conf-of-interest}(\mathsf{p},\text{user}) \\ \wedge \exists \mathsf{d} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{d}) \end{pmatrix} \Bigg) \Bigg)
\end{aligned}
$$

Note that we don't need to check that user! = d and ¬Author(user) to prevent the author from submitting a review for the paper given that she has to be assigned to be the reviewer of that paper. In EasyChair, the Chair or the PC-members are expected to declare conflict of interest by themselves manually which prevent the PC member from becoming reviewers of their own paper. This is included in the rule as we check for conflict of interest.

2- When the review menu is enabled and the submitted paper is not deleted: **(a)** A PC chair can submit a review for any paper as himself. **(b)** A PC chair can submit a review for a paper as another PC member using "log in as another pc member" if the PC member is allowed to submit a review for that paper. **(c)** A PC member can review a paper if she is assigned to review that paper. **(d)** A PC member can review a paper to which she is not assigned when PC members are permitted to access the titles and reviews of submitted papers. She also must have no conflict of interest with that paper.

Action Add-Review(p,a,b):-

{

Submitted-review(p,a,b):= ⊤;

}

$$\text{exec}(\text{Add-Review}(\mathsf{p},\mathsf{a},\mathsf{b}),\text{user})) \rightleftharpoons$$

$$\left(\begin{array}{l} \left(\begin{array}{c}\text{Chair}(\text{user}) \wedge \text{Chair-review-en}() \\ \wedge \mathsf{a} = \text{user} \wedge \exists \mathsf{d} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{d})\end{array}\right) \\ \vee \left(\begin{array}{c}(\mathsf{a} = \text{user} \vee \text{Chair}(\text{user})) \\ \wedge \exists \mathsf{c} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{c}) \\ \wedge \left(\begin{array}{c}\left(\begin{array}{c}\text{PCmember}(\mathsf{a}) \wedge \text{Reviewer}(\mathsf{p},\mathsf{a}) \\ \wedge \text{PCM-review-menu-en}() \\ \wedge \text{PCM-review-editing-en}()\end{array}\right) \\ \vee \left(\begin{array}{c}\text{PCmember}(\mathsf{a}) \wedge \neg\text{Reviewer}(\mathsf{p},\mathsf{a}) \\ \wedge \text{PCM-review-menu-en}() \\ \wedge \text{View-sub-title-permitted}() \\ \wedge \text{PCM-access-reviews-en}() \\ \wedge \neg\text{Conf-of-interest}(\mathsf{p},\mathsf{a})\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

3- Given that paper assignments are enabled, a PC chair can assign/de-assign a submitted paper to a PC member or a PC chair for reviewing, when she has no conflict of interest with that paper.

Action Add-Reviewer-Assignment(p,a)

{

Reviewer(p,a) := $\top$;

}

$$\text{exec}(\text{Add-Reviewer-Assignment}(\mathsf{p},\mathsf{a}),\text{user}) \rightleftharpoons \left(\begin{array}{c}\text{Chair}(\text{user}) \wedge (\text{PCmember}(\mathsf{a}) \vee \text{Chair}(\mathsf{a})) \\ \wedge \text{Review-assig-enabled}() \\ \wedge \neg\text{Conf-of-interest}(\mathsf{p},\mathsf{a}) \\ \wedge \exists \mathsf{c} : \text{Agent} \ . \ \text{Submitted-paper}(\mathsf{p},\mathsf{c})\end{array}\right)$$

4- When the review menu is enabled and the submitted paper is not deleted: **(a)** A PC chair can request another agent to subreview any paper. **(b)** A PC member can invite another agent to subreview a paper: (1) if she is the reviewer of the paper or (2) if the system is configured to give PC members access to the paper submission titles and reviews. The invited agent can decide whether to accept or

reject the reviewing request as long as the paper has not been withdrawn. A PC member cannot cancel the subreviewing request but can accept or reject the request on behalf of the invited agent. Once the decision is made, only the PC member can change the decision.

Action Request-Reviewing(p,a,b)

{

Requested-subrev(p,a,b):= $\top$;

}

Action Accept-Reviewing-Request(p,a,b)

{

Decided-subrev(p,a,b):=$\top$;

Subreviewer(p,a,b):=$\top$;

}

Action Reject-Reviewing-Request(p,a,b)

{

Decided-subrev(p,a,b):=$\top$;

Subreviewer(p,a,b):=$\bot$;

}

$$\text{exec}(\text{Reject-Reviewing-Request}(\text{p}, \text{a}, \text{b}), \text{user}) \rightleftharpoons \left(\begin{array}{c} \text{Requested-subrev}(\text{p}, \text{a}, \text{b}) \\ \wedge \exists \text{c} : \text{Agent} \ . \ \text{Submitted-paper}(\text{p}, \text{c}) \\ \wedge \left(\ \neg \text{Decided-subrev}(\text{p}, \text{a}, \text{user}) \vee \text{user} = \text{a} \ \right) \end{array} \right)$$

$$\text{exec}(\text{Accept-Reviewing-Request}(\text{p}, \text{a}, \text{b}), \text{user}) \rightleftharpoons \left(\begin{array}{c} \text{Requested-subrev}(\text{p}, \text{a}, \text{b}) \\ \wedge \exists \text{c} : \text{Agent} \ . \ \text{Submitted-paper}(\text{p}, \text{c}) \\ \wedge \left(\ \neg \text{Decided-subrev}(\text{p}, \text{a}, \text{user}) \vee \text{user} = \text{a} \ \right) \end{array} \right)$$

$$\text{exec}(\mathsf{Request\text{-}Reviewing(p,a,b)}, \mathsf{user}) \rightleftharpoons \left(\begin{array}{l} \left(\begin{array}{c} \mathsf{Chair\text{-}review\text{-}en()} \wedge \mathsf{Chair(user)} \\ \wedge \exists \mathsf{c} : \mathsf{Agent}\ .\ \mathsf{Submitted\text{-}paper(p,c)} \end{array} \right) \\ \vee \left(\begin{array}{c} \mathsf{PCmember(user)} \wedge \mathsf{Reviewer(p,user)} \\ \wedge \mathsf{PCM\text{-}review\text{-}menu\text{-}en()} \\ \wedge \exists \mathsf{c} : \mathsf{Agent}\ .\ \mathsf{Submitted\text{-}paper(p,c)} \end{array} \right) \\ \vee \left(\begin{array}{c} \mathsf{PCmember(user)} \wedge \neg \mathsf{Reviewer(p,user)} \\ \wedge \mathsf{PCM\text{-}review\text{-}menu\text{-}en()} \\ \wedge \mathsf{View\text{-}sub\text{-}title\text{-}permitted()} \\ \wedge \mathsf{PCM\text{-}access\text{-}reviews\text{-}en()} \\ \wedge \neg \mathsf{Conf\text{-}of\text{-}interest(p,user)} \\ \wedge \exists \mathsf{c} : \mathsf{Agent}\ .\ \mathsf{Submitted\text{-}paper(p,c)} \end{array} \right) \end{array} \right)$$

We include the full model of ***EC*** at [90].

5.4 Analysis of *EC* security properties using *X-Policy*

In this Section, we will present three security properties in ***EC***. We have discovered these issues while using EasyChair. In each case, we show an attack strategy to achieve an undesirable state. These attacks are derived using ***EC*** and *X-Policy* and have succeeded on EasyChair as of 1st of September 2009. Perior to the publication of these attacks, the developers and administrators of EasyChair were notified about the discussed attacks on the 10th of April 2009 and on the 1st May 2009. Despite our best effort to contact them, no response has been given. We are unable to repeat these attacks given that experiments on the live system is prohibited in the current license.

In the following, we report the results of each attack and make some suggestions on how to fix the system. During the process of analysing ***EC*** a number of other properties that holds in ***EC***. For example, ***EC*** satisfy the property that a Chair cannot assign a PC-member to review a paper if she has a conflict of interest with it. Another property that ***EC*** holds is the fact that a reviewer cannot read another reviewer's review of a paper assigned to her before she submits her review of the paper if the Chair choose to restrict

the viewing of the paper submissions to the relevant PC-members.

For our ***EC*** model, we create the following configuration:

1. The system has five agents: Alice, Bob, Eve, Carol and Marvin. The system has two submitted papers: p1 and p2.

2. Alice is the Chair of PC. Bob and Carol are PC members. Paper p1 is submitted by the author Marvin while p2 is submitted by the author Eve.

The configuration and the attacks' detailed derivation of the model ***EC*** has been discussed in [92]. We also describe the *X-Policy* queries that represent these properties. For space restriction, readability and presentation purposes, we describe an optimal strategy that explains the attack for each property as the tool-generated strategies can be complex.

5.4.1 Property 1: A single subreviewer should not be able to determine the outcome of a paper reviewing process by writing two reviews of the same paper.

We show that we can derive an attack against ***EC*** involving 4 agents: Alice, Bob, Carol, and Eve. We explain the attack scenario as a sequence of actions executed by these agents as follows:

1. Alice acts as chair. She executes the actions: Add-Reviewer-Assignment(p1,Bob) to assign Bob to review the paper p1. She also executes Add-Reviewer-Assignment(p1,Carol) to assign Carol to review the paper p1.

2. Bob and Carol both assign Eve as their sub-reviewer for paper p1 by executing the actions Request-Reviewing(p1,Bob,Eve) and Request-Reviewing(p1,Carol,Eve) respectively.

3. Eve accepts the two paper subreviewing requests and sends Bob and Carol two similar reviews using Accept-Reviewing-Request(p1,Carol,Eve) and Accept-Reviewing-Request(p1,Bob,Eve).

4. Bob and Carol receive Eve's reviews and submit them to the system using Add-Review(p1,Bob,Eve) and Add-Review(p1,Carol,Eve).

X-Policy query that represent this property is:

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)!
        and Author(p1,Marvin)! and Author(p2,Eve)!
        and PCmember-review-editing-enabled()!
        and View-submission-by-chair-permitted()!
        and Chair-review-menu-enabled()!
        and PCmember-review-menu-enabled()!
        and Submission-anonymous()!
        and Review-Assignment-enabled()!
              -> {Alice,Carol,Bob}:( {Submitted-review(p1,Bob,Eve)}
                    THEN {Alice,Carol,Bob}:{Submitted-review(p1,Carol,
                      Eve)})}
```

EasyChair fails this property and allows Eve to submit two reviews for the same paper. One possible fix for this attack is as follows. Every time an agent a invites another agent b to subreview a paper, EasyChair should check whether agent b has been invited by another agent to subreview the same paper. We conjoin the condition $\neg\exists$ d : Agent . Requested-subrev(p, d, b) to the permission statement exec(Request-Reviewing(p,a,b),user). Carol cannot execute Request-Reviewing(p1,Carol,Eve) as Requested-subrev(p1,Bob,Eve) is in the previous state.

5.4.2 Property 2: A paper author should not review her own paper.

As before, we explain the attack scenario as a sequence of actions executed by the agents Alice, Bob and Eve:

1. Alice acts as Chair and assigns Bob, who is a PC member, to review the paper p2 submitted by Eve by executing the action Add-Reviewer-Assignment(p2,Bob).

2. Bob executes the action Request-Reviewing(p2,Bob,Eve) to assign Eve as his sub-reviewer as she is a good researcher in the field.

3. Eve accepts the request using Accept-Reviewing-Request(p2,Bob,Eve).

4. Bob submits the review using Add-Review(p2,Bob,Eve).

We represent this property in *X-Policy* as the following check statement

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
  Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)!
        and      Author(p1,Marvin)! and Author(p2,Eve)!
        and PCmember-review-editing-enabled()!
        and View-submission-by-chair-permitted()!
        and Chair-review-menu-enabled()!
        and PCmember-review-menu-enabled()!
        and Submission-anonymous()!
        and  Review-Assignment-enabled()!
                        -> {Alice,Carol,Bob,}: {Submitted-review(p2,Bob,Eve
                            )} }
```

In this case, EasyChair fails the property and allows Eve to review her own paper. Note that the names of the authors and other reviewers are not known to the PC members.

One possible fix for this attack is that every time an agent a invites another agent b to subreview a paper, EasyChair should check whether agent b is actually an author of that paper. We add the condition ¬Author(p, a) to the permission statement exec(Request-Reviewing(p,a,b),user). In this case Bob cannot execute Request-Reviewing(p2,Bob,Eve).

5.4.3 Property 3: Users should be accountable for their actions.

This property is violated in several ways, all of which involve the use of "log in as another pc member". For example, the system should not allow the chair to submit a review for a paper as another PC member without making it clear that it is actually the chair who has submitted the review and not the PC member. The following attack scenario involves Alice and Bob:

1. Alice is the chair. She executes Add-Reviewer-Assignment(p1,Bob) to assign Bob to review the paper p1.

2. Bob submits his review using Add-Review(p1,Bob,Bob).

3. Alice reads Bob's review of paper p1 by executing Show-Review(p1,Bob,Bob).

4. Alice submits a review for the paper p1 as if she is Carol who is a very famous and sought after academic by executing Add-Review(p1,Carol,Carol).

We represent this property in *X-Policy* as the following check statement

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)!
        and Author(p1,Marvin)! and Author(p2,Eve)!
        and PCmember-review-editing-enabled()!
        and View-submission-by-chair-permitted()!
        and Chair-review-menu-enabled()!
        and PCmember-review-menu-enabled()!
```

```
and Submission-anonymous()!
and Review-Assignment-enabled()!
        -> {Alice,Bob}: {Submitted-review(p1,Carol,Carol)} }
```

Note that Carol is not part of the coalition. EasyChair fails this property and allows the chair to read another reviewer's review for a paper and then submits a review for that paper as another PC member without being detected by the other PC members or the other chairs. This attack is possible because the system does not register the name of the user who updated the review. It will appear to others as if Carol has submitted the review herself. One possible fix for this attack is for Add-Review() to have an additional parameter. Alice would then need to execute the action Add-Review(p,a,b,c) where agent a is the chair acting on behalf of b who is the PCmember submitting the review written by agent c. The predicate Submitted-review() also has to be changed accordingly.

One of the strategies output by *X-Policy* for the ***EC*** model verifying the property no. 3 is available in the Listing 5.1. The full output file is available online at the URL: http://www.cs.bham.ac.uk/~hxq/xpolicy/ec.x.

```
Assignment: [p1=1 p2=2 Alice=1 Carol=2 Bob=3 Marvin=4 Eve=5]

Strategy: 1.3
Coalition: [1, 3]
Execute the action AddReview(1,2,2)  by agent 1;
skip;

The goal is reachable. Number of Strategy found: 1
```

Listing 5.1: The *X-Policy* output file for ***EC*** file verifying property no. 3.

5.5 Chapter Summary

Chapter 5 presented the process of modelling and verifying the access control policy of EasyChair conference management system. Section 5.1 provided some background

information of conference management systems and overview of the conventions used in the constructing the model is discussed in Section 5.2. The description of ***EC*** access control policy in *X-Policy* is discussed in Section 5.3. Section 5.4 provided an analysis of number of security properties and suggested a number of changes to the access control policy when appropriate. In Chapter 6 we evaluate the software implementation of *X-Policy* tool.

CHAPTER 6

X-Policy Tool: Implementation and Evaluation

Chapter 3 introduced *X-Policy* modelling language for modelling dynamic access control policies with compound actions. *X-Policy* allow us to express access control policies by building a model based on propositional variable. Users of *X-Policy* can then express security properties they want to verify using *X-Policy* query language. In Chapter 4, we discussed the model checking algorithm and how we analyse *X-Policy* queries using user knowledge and how the algorithm searches for strategies. We also discussed the agents' knowledge representation.

In this Chapter we present the implementation of *X-Policy* tool. We start by describing the structure of the tool and the its main components in Section 6.1 which also discuss languages and packages used for implementing various components. Section 6.2 explains the tool usage usage information. The source code structure and overview are discussed in Section 6.3. The semantic checking implementation is discussed and the detailed semantics rules applied to *X-Policy* script are listed in Section 6.4. We explain the notion of computational rounds in Section 6.5.

In Section 6.6, we evaluate the *X-Policy* framework. In Section 6.6.1 we introduce a running example where we explore the tool working from encoding the policy into *X-*

Policy code to generating the strategy and we explain how to use the resulted strategy to fix the policy. In Section 6.6.2, the *X-Policy* tool as a product is evaluated and the performance analysis of the tool is analysed with a comparison against similar tools where possible. We summarise this chapter in Section 6.7.

This Chapter is designed to enable and facilitate the use and the incremental development of the software tool. The scope of this chapter covers the tool design and implementation including the tool performance, results and examples.

6.1 Implementation Overview

X-Policy tool is a Java-based open-source software implementation of the *X-Policy* model checking algorithm. The tool allows the user to input a policy model written in *X-Policy* modelling language as a script file which must also contain the security property (goal) written in *X-Policy* query the users wish to verify. The tool, then, carries out the parsing and the syntax and semantic checking on the *X-Policy* script file as we can see in Figure 6.1. During this step input data needed for the model checking algorithm is collected. The model checking is then curried out and the tool then outputs strategies if found. During these various steps if an error occurs (e. g. syntax or semantics error), the software may

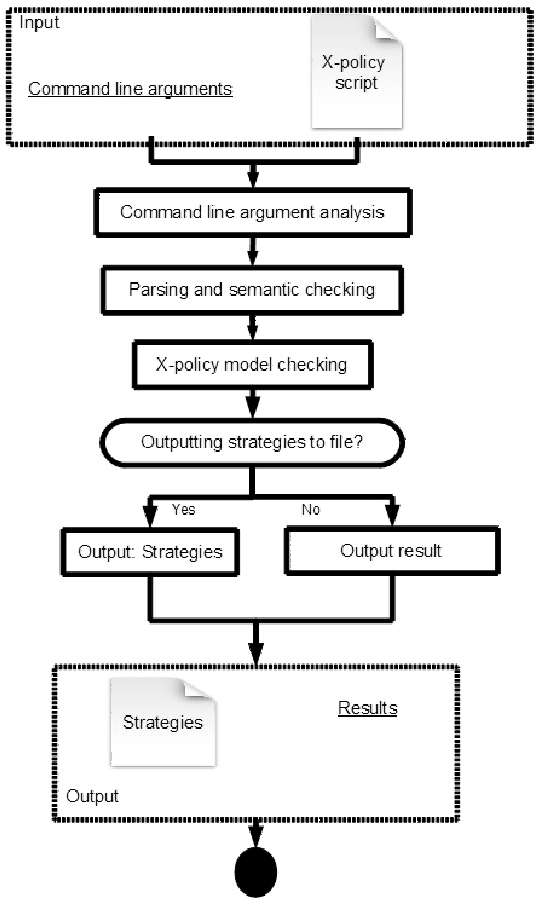

Figure 6.1: The X-policy working flow-chart.

throw an error and terminate.

The *X-Policy* tool is written in Java.

The *X-Policy* parser is written using JavaCC [57] and JJTree [29]. The *X-Policy* model checking algorithm handles Binary Decision Diagram operations using JavaBDD package [118]. JavaBDD is a Java library for manipulating BDDs which provides an interface for the BDD native packages written in C: Buddy [28] and Cudd [107].

The source code, documentation and executable builds are available for download on-line from the author website[1]. JavaBDD binaries and Buddy libraries for Windows and Linux are included in *X-Policy* package. As JavaBDD uses some native packages for CUDD and Buddy, *X-Policy* inherently has platform dependency. *X-Policy* is run and tested using Java 1.6, JavaCC 5 and JavaBDD 1.0. *X-Policy* executable builds are available for Windows and Linux. The coding style used follows the guidelines summarised in [104].

6.2 Usage Information

This section illustrate the usage information for the tool. *X-Policy* is available for execution as a JAR file: `xpolicy.jar`. For example, the command below will make the tool model check the script `conference` and output any strategy found to the file `output/strategy.acc` in current folder.

```
java -jar xpolicy 1rp examples/conference.x output/strategy.acc    (1)
```

In the command line as in Command 1, the position between `java -jar xpolicy` and the script path and file name `examples/conference.x` is reserved for arguments that will determine the behavior of *X-Policy*.

We summarise the usage of these arguments in the following list.

[1]The tool is available at: `http://www.cs.bham.ac.uk/~hxq/xpolicy`

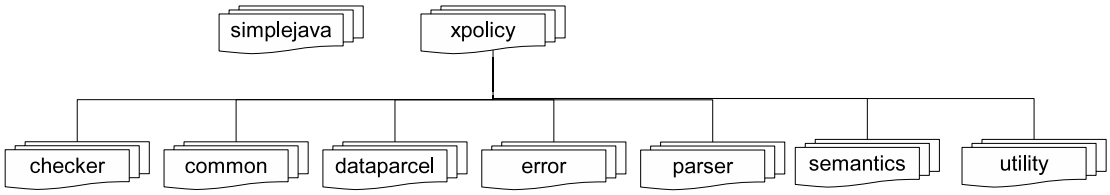

Figure 6.2: The structure of the *X-Policy* package.

0/1 The argument '0' is for telling *X-Policy* to run using the algorithm Algo-0, whereas '1' is for telling *X-Policy* to run using the algorithm Algo-1. They are mutual-exclusive in the command line. Algo-0 is the algorithm discussed in the section 4.4.3 while Algo-1 is an alternative algorithm to Alog-0 which we discuss in Appendix A where we also explain the different way each Algorithm handle newly found states.

p It is used to tell *X-Policy* to output the strategies found during the checking. If it is presented in the argument list, before each round of a checking starts, *X-Policy* prompts a question, asking whether strategies found in this round should be output. If it is absent, *X-Policy* does not prompt the question at the beginning of each round, but only returns an answer "yes" or "no" for this round of checking. The concept of rounds is discussed in Section 6.5

r It is used to tell *X-Policy* to run every round of a checking. If it is absent, *X-Policy* prompts a question before the starting of each round, asking whether this round should be running.

6.3 Source-code package structure

X-Policy Java package, as illustrated in figure 6.2, contains two components:

- `SimpleJava`, a Java package that deals with input and output, written by Jim McGregor, developed at School of Computer Science, University of Birmingham.

- `xpolicy` which is the top-level package and contains a number of sub-packages to handle the various aspect of *X-Policy* implementation:

  `xpolicy.parser` which contains classes to process the command line arguments as discussed in the previous Section and *X-Policy* compiler files generated by JavaCC and JJTree.

  `xpolicy.semantics` which contains the methods for semantics checking which is detailed in Section 6.4.

  `xpolicy.error` which contains the classes that handle the various exceptions thrown out when errors occur during the execution of the tool.

  `xpolicy.common` which contains a number of constants and variables that are accessed commonly by more than one class.

  `xpolicy.dataparcel` which contains the classes used to represent various data structures like Predicate, Parameter, Goal Coalition pairs.

  `xpolicy.utility` which provides a number of classes to handle and manipulate data structures used by other classes like: arraylists, parsed-trees and BDD-trees.

  `xpolicy.checker` that contains the implementation of the model checking algorithm.

6.4 The semantic checking

6.4.1 The three methods in `xpolicy.semantic.SemanticChecker`

The semantic checking proceeds side by side with the parsing. This is made possible by JavaCC. It allows the Java code to be embedded throughout the syntax definition in the JJTree script. When a token is (about to be) read from the inputting stream, the

appropriate methods in `xpolicy.semantic.SemanticChecker` are invoked to perform relevant processing or checking on the token. There are three major methods in the class for these purposes. They are:

- `onEnterChecking(...)`, doing preparing work before a token is checked, such as initialising variables;
- `checkToken(...)`, checking a token against certain semantic rules after it is read;
- `onExitChecking(...)`, checking a token against certain semantic rules when the parser is about to finish parsing the grammatical unit that contains the token.

The structure of the three methods are very similar. In `checkToken(...)` and `onExitChecking(...)`, the code for checking are wrapped by the `try` block of a `try-catch` statement. If any semantic error is found by the code in the `try` block, an exception of the type `xpolicy.error.XpolicyException` is thrown out and subsequently caught by the code in the `catch` block. After that, error messages are printed out on the screen and the program terminates. Inside the `try` block there is a `switch` statement. Each `case` branch of the `switch` statement is an entrance point for the block of code that is used to check certain grammatical unit of *X-Policy* language. Whenever the methods are invoked from `xpolicy.parser.XPparser`, an `int` variable is passed to the methods as one of the arguments, whose value is used for picking an appropriate `case` branch in the method invoked. In `onEnterChecking(...)`, there is no such `try-catch` statement, but only a `switch` statement like the other two methods.

The three classes, `xpolicy.parser.XPparser`, `xpolicy.semantic.SemanticChecker`, and `xpolicy.error.XpolicyException`, interact when preforming the semantic checking on an *X-Policy* script. Using an example Figure 6.3 illustrates that.

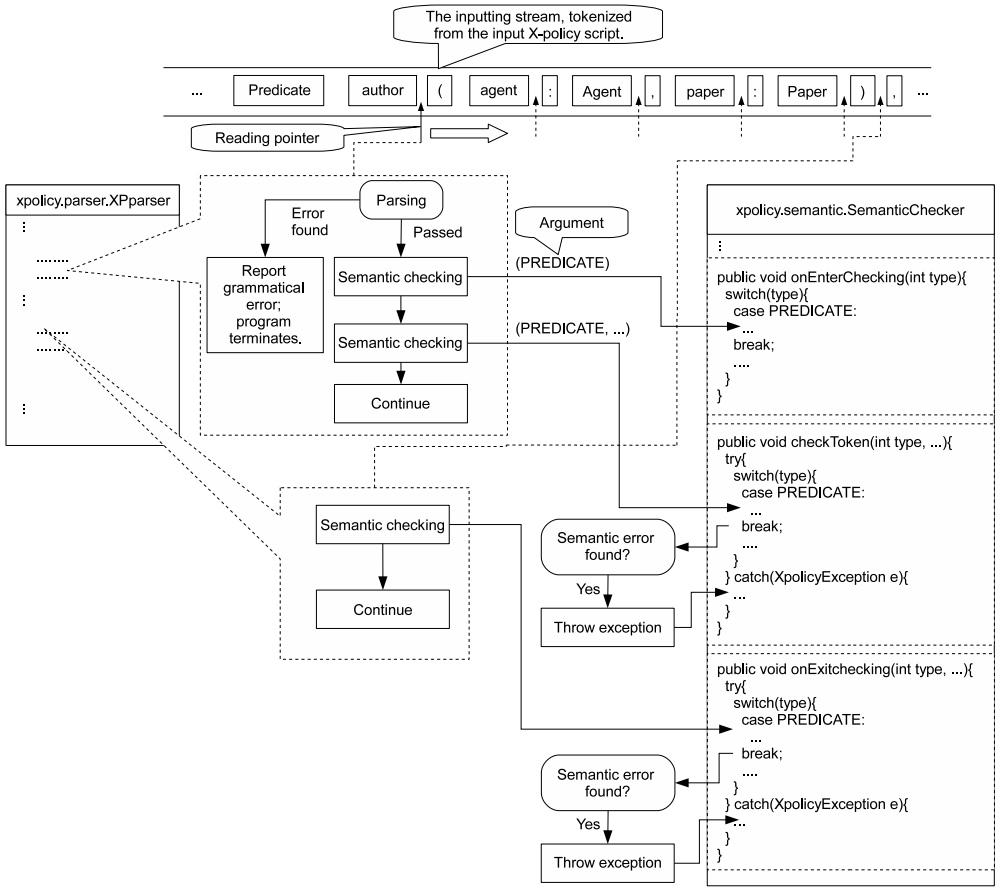

Figure 6.3: A working flow illustration about the three classes when doing the semantic checking.

6.4.2 Rules applied in the semantic checking and some implementation notes

6.4.2.1 Names for the grammatical units

Before writing down the semantic rules applied to the semantic checking, we shall properly name the grammatical units that comprise an *X-Policy* script so that we can refer to them later. The names are summarised in Figure 6.4.

6.4.2.2 Semantic rules and some implementation notes

In *type definition section*:

- *Type name* must be unique.
- *Type name* must start with an upper case letter.
- The type `Agent` is pre-defined.

In *predicate definition section*:

- *Predicate name* must be unique.
- *Parameter name* must start with a lower case letter.
- *Parameter type* must be one of the defined *type name*s.
- *Parameter name* in a *predicate definition* must be unique.

Note:

- If a *predicate definition* is marked by '!' at the end, the predicate becomes a *constant predicate*. Only one propositional variable among all propositional variables built from such a predicate can have the boolean value *true*. The user should explicitly

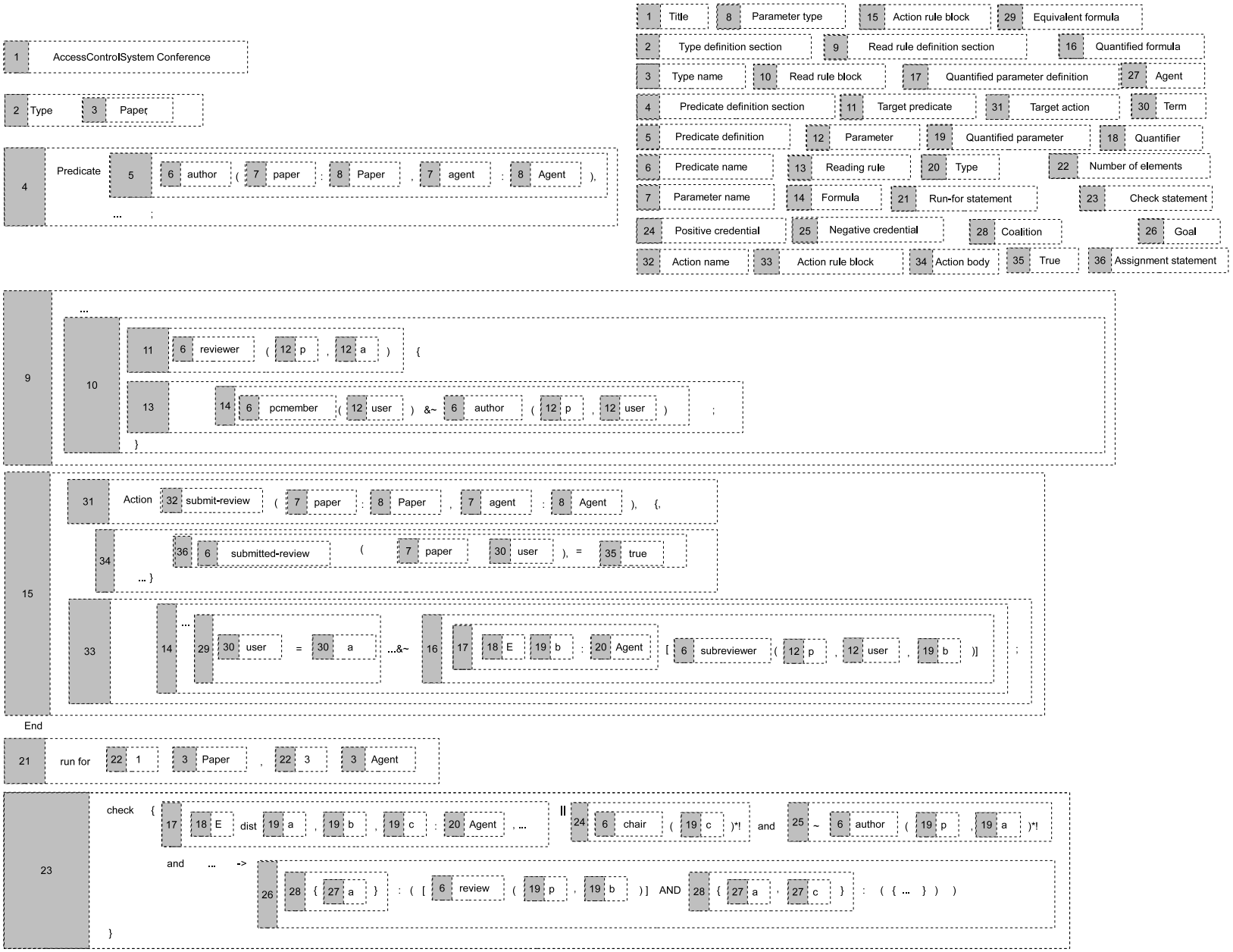

Figure 6.4: Names for the grammatical units that comprise an *X-Policy* script.

point out the variable whose value should be *true* by using a *positive credential* in the *check statement*.

On the *target predicate* of a *read rule block*:

- *Predicate name* must have already been defined in the *predicate definition section*.
- The number of the *parameter*s must agree with the number of parameters of the defined predicate.
- *Parameter*s may not be unique.

On the *target action* of an *action rule block*:

- *Parameter name* must start with a lower case letter.
- *Parameter type* must be one of the defined *type name*s.
- The pair of the *action name* and the parameter list must be unique.

Notes:

- At this point, in `xpolicy.semantic.SemanticChecker`, a parameter list is created. The list will not be created again until the parser comes out of the *rule block*. The list is for storing *parameter*s and later for retrieving.
- At this point, each *parameter* is assigned an appropriate type according to the *parameter type* that appears at the parameter's position in the *predicate definition*.

This applies to any rule block whether it is an action rule or a reading rule. In addition, in a rule block:
For an *Action body*:

- *Predicate name* must have already been defined in the *predicate definition section*.

- The number of the *parameter*s must agree with the number of parameters of the defined predicate.
- *Parameter*s may not be unique.
- All *parameter*s appearing in the action body must be defined in the surrounding for-loop or in the target action definition.
- *parameter*s declared in for-loop must not been used before in enclosing for-loop, user or the action parameters.
- The *parameter* `user` is pre-defined.

For an *Action rule block* and *Read rule block*:

- The number of *parameter*s of a predicate must agree with the number of the defined predicate.
- The *parameter* `user` is pre-defined.
- *Parameter* other than `agent` must already been defined, either as one of the *parameter*s of the *target predicate/action*, or in a *quantified parameter definition*.
- The type of a *parameter* must agree with the type of the position where it appears.
- Both *term*s of an *equivalent formula* must already been defined.
- Both *term*s of an *equivalent formula* must be of the same type.
- In a *quantified parameter definition*, each *quantified parameter* must not have been defined already.
- The *type* in a *quantified parameter definition* must be one of the defined *type name*s.

- A *quantified formula* creates its own name space. *Quantified parameter*s defined inside are invisible from outside.

For *run-for statement*:

- *Type name* must have already been defined in the *type definition section*.
- One *type name* can only be populated once. (The term *populate* refers to the assigning of elements to a type, here, denoted by a *type name*.)
- All the defined *type name*s must be populated.

In *check statement*:

- All *parameter*s appearing in credentials and in *goal* must be defined in *quantified parameter definition*.
- For *quantified parameter definition*, the same semantic rules apply as for *quantified parameter definition* in *formula*.
- For distinct *parameter*s defined on a *type*, their number must be no greater than the number of elements assigned to the *type* in the *run-for statement*.
- For *positive credential*s and *negative credential*s, the same semantic rules apply as for parameterised predicates in *formula*.
- For parametrised predicates appearing in *goal*, the same semantic rules apply as for parameterised predicates in *formula*.
- In a *coalition*, an *agent* must have already been defined in the *quantified parameter definition* as an `Agent`. An *agent* cannot appear more than once in a *coalition*.

6.5 Computation rounds

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 the following:

```
check {E dist a,c:Agent,p:Paper|| chair(c)*!->{c}:{reviewer(p,a)}}    (2)
```

preceded by the run statement:

```
run for 3 Paper, 4 Agent    (3)
```

there are three quantified variables: a, c (disjointed) are defined on the set `Agent`, and p is defined on the set `Paper`. The set `Agent` is populated to four elements – $\{a_1, a_2, a_3, a_4\}$ – and the set `Paper` to three elements – $\{p_1, p_2, p_3\}$ – by the execution of the run-statement. Therefore the possible values that a can have is four, the possible values for c is four, and the possible values for p is three. The total number of combinations of the values of a, c and p is forty-eight. Each of the combinations, if run by the algorithm, becomes a round.

However, to obtain the overall result for checking a query, not every round is executed by the tool. The use of the keyword `dist` 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.

However, the computation can be further simplified. In the cases that all quantified variables are made distinct using the keyword `dist`, 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.

Alternatively, we can write queries where we would like to consider situations where c and a refer to the same object as in the following query:

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

When running the tool, we leave the decision of running which round to the user as discussed in Section 6.2 . In the following discussions, all the experimental results on computational time and memory usage are the results obtained from running just one round.

6.6 Evaluation

In Section 6.6.1 we introduce a running example where we explore the tool working from encoding the policy into *X-Policy* code to generating the strategy and we explain how to use the resulted strategy to fix the policy. The *X-Policy* tool as a product is evaluated and the performance analysis of the tool is analysed with a comparison against similar tools where possible.

6.6.1 On the end product: a running example

In this Section, we introduce a running example to evaluate the tool. We start by discussing the construction of the input file and we then illustrate the tool running as well as its output. We will use a running example (based loosely on the example discussed in 2.3.1) to illustrate the working of *X-Policy* and explain the input and output of the tool and how we can interpret these results.

In Listing 6.1 we explore a simple example where there are a number of variables, `x(p)`,

```
AccessControlSystem xyuz_example
        Type P;

        Predicate u(p: P), x(p: P), y(p: P),z(p: P);

        u(p){read: y(p);}
        x(p){read: true;}
        y(p){read: true;}
        z(p){read: x(p) or ~y(p);}

        Action X2T(p:P) {x(p):=true;}{~u(p);}
        Action X2F(p:P) {x(p):=false;}{~u(p);}
        Action Y2T(p:P) {y(p):=true;}{u(p);}
        Action Y2F(p:P) {y(p):=false;}{u(p);}
        Action U2T(p:P) {u(p):=true;}{x(p);}
        Action U2F(p:P) {u(p):=false;}{ true;}
        Action Z2T(p:P) {z(p):=true;}{x(p) or y(p);}
        Action Z2F(p:P) {z(p):=false;}{x(p) or y(p);}
End
run for 1 P, 1 Agent
check{E p: P, a: Agent || {a}:({~u(p)} AND {a}:([z(p)] ))}
```

Listing 6.1: The *X-Policy* input file for the XYUZ example.

`y(p)`, `u(p)` and `z(p)` on which the user can read and perform actions. The following *X-Policy* code specifies the actions and read and execution permissions. In the Query we ask whether or not there is a strategy for an agent `a` to execute so that he can reach the goal of changing the value of `u(p)` to false and read the value of `z(p)`.

We then run the *X-Policy* tool giving the file in the Listing 6.1 as an input. The tool performs the model checking algorithm and outputs, in this case, a number of strategies. We include one of these strategies in Listing 6.2 where the agent has built her knowledge of the system by reading the model variables and decided which actions to execute based on the value of these variables. This can be useful for analysing and determining the dependencies between variables in the access control policy. The next Section evaluates the *X-Policy* tool against similar tools. We discuss our methodology, experiments, results and the examples we used in the evaluation.

```
Strategy: 1.1
Coalition: [1]
if (y(1) is true) by 1 {
   if (u(1) is true) by 1 {
      Execute the action Y2F(1)  by agent 1;
      if (z(1) is true) by 1 {
         Execute the action U2F(1)  by agent 1;
         skip;
      }else {
         Execute the action U2F(1)  by agent 1;
         skip;
      }
   }else {
      Execute the action X2T(1)  by agent 1;
      if (z(1) is true) by 1 {
         skip;
      }else {
         skip;
      }
   }
}else {
   if (z(1) is true) by 1 {
      Execute the action U2F(1)  by agent 1;
      skip;
   }else {
      Execute the action U2F(1)  by agent 1;
      skip;
   }
}
The goal is reachable. Number of Strategy found: 1
```

Listing 6.2: The *X-Policy* output file for the XYUZ example.

6.6.2 Evaluation of *X-Policy* against similar tools.

In this Section, we evaluate *X-Policy* against similar tools that have been used to model and analyse dynamic access control policies. This section also discusses issues resulting from the translation process. We evaluate a number of access control systems with a number of properties. In Section 6.6.3 we evaluate *X-Policy* against *RW* (*X-Policy* precursor) and DyNPAL runtime performance.

During this evaluation we will use a number of access control systems first introduced

by [121] and later used for evaluating DyNPAL in [12], as discussed in Section 6.6.3:

Employee information system (EIS) is used to enforce authorisation rules on bonus allocation among the employees of a department. A bonus package with a fixed number of options, such as a free day off work, is available for all employees. The director of the department chooses options from the package to give to all employees. She can also read the information about the distribution of options. The director can promote an employee to be a manager. Managers can read and set ordinary employees' bonuses, but not bonuses of other managers or the director. An employee can appoint another employee to be his advocate, and have read access to his bonus information – for example, this might be useful if he needs help from a trade union. We detail this model in *X-Policy* in the Section D.1.

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.

We include the SIS model in *X-Policy* in Section D.2.

Conference review system (CRS) which describes the access control policy regarding the process of submitting and reviewing papers in a small conference which is encoded in the *X-Policy* script in Section D.3. This is different from ***EC*** which we discussed in Chapter 5 as *RW* does not support modelling of compound actions.

6.6.3 Performance analysis of *X-Policy*

In this section we compare the performance analysis of *X-Policy* against similar tools. We use a version of *RW* that does not support guessing strategies, and DYNPAL. For DYNPAL, we use the performance data provided in [12] as the tool is not available for public use. The queries described in Section 5.4 cannot be verified by *RW* as it does not support compound actions. However, in order to compare performance, we take 4 queries from [121] that can be handled by both tools and DYNPAL. Note that DYNPAL differs from *RW* and *X-Policy* as it does not consider an agent's knowledge in its system representation, search algorithm and state transitions. This table demonstrates that while the *X-Policy* tool uses Binary Decision Diagrams which are prone to state explosion problem, it still manages to have a reasonable evaluation time. The query evaluation time for *RW* is different from those in [121, 12] as we are using a version of *RW* that does not support guessing strategies. In order to compare the performance of *X-Policy* against *RW* , the access control policies written in *RW* must be translated to *X-Policy*. Each of the overwriting rules in *RW* is mapped to their equivalent action rules. Each variable over-writing rule is split into two actions (*write to true* and *write to false*) with the appropriate permission statement based on the access policy similar to the example discussed in Section 6.6.1 which is a corresponding model the RW model discussed in Section 2.3.1.

run run for 3 **Paper**, 4 **Agent**

```
check {E dist a,c:Agent, p:Paper ||
        chair(c)*!
                ->{c}:{reviewer(p,a)}}
```

Listing 6.3: *X-Policy* Query 4.2 for conference review system.

We run the Query 4.2 which is expressed using *X-Policy* in Listing 6.3 for the CRS with 7 objects (3 papers and 4 agents that looks for strategies which an agent can promote herself to become a reviewer of a paper.

```
run run for 3 Paper, 4 Agent
check {E dist a,b,c: Agent, p: Paper ||
        chair(c)*! & ~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)]
                THEN {a,c}:({submittedreview(p,a)}))}
```

Listing 6.4: *X-Policy* Query 4.3 for conference review system.

Query 4.3 as expressed in Listing 6.4 for CRS which is a nested query that asks if a reviewer can submit her review for a paper while she has read the review of someone else before.

```
run for 3 Bonus, 3 Agent
check{E dist a1,a2: Agent, b: Bonus ||
        ~director(a1)*! and ~director(a2)*!
        and manager(a1)! and manager(a2)!
        and ~bonus(a1,b)!
                -> {a1,a2}:({bonus(a1,b)})}
```

Listing 6.5: *X-Policy* Query 6.4 for EIS system.

Query 6.4 as in Listing 6.5 with 6 objects for EIS which evaluates if two managers can collaborate to set a bonus for one of them and Query 6.8 as in Listing 6.6 for SIS with 10 objects that asks if a lecturer can assign two students as the demonstrator of each other.

```
run for 10 Agent
check{E dist l, a1, a2: Agent ||
        lecturer(l)*! and student(a1)*!
        and student(a2)*! and higher(a1,a2)*!
                -> {l}:{demonstrator_of(a1,a2) and demonstrator_of(a2,a1)}}
```

Listing 6.6: *X-Policy* Query 6.6 for SIS system.

These examples cover a number of scenarios and allow us to express and verify a number of properties concerning each policy.

Figure 6.1 shows a reduction in time by the *X-Policy* algorithm compared RW. As expected, the verification time[1] and state space grow exponentially when more objects are added. However, *X-Policy* seems to perform better than *RW* in that sense. *X-Policy* seems to perform the model checking algorithm in a reasonably acceptable time. In Table

Table 6.1: Query evaluation time in s.

Query(from [121])	***X-Policy***	**DYNPAL**	*RW*
Query 4.2	0.254	0.120	0.447
Query 4.3	0.286	0.125	0.782
Query 6.4	0.780	0.120	0.892
Query 6.8	0.360	0.120	0.945

6.3, we record the number of variables, performance time for *X-Policy* and *RW* in relation to the number of agents and papers for the Query 4.4 of the CRS with five-level nested queries that checks if an agents can be assigned as a pcmember by the chair and then resign her membership. As expected, we can see in Figure 6.5 that the number of variables

[1]total time for the rounds

Table 6.2: *X-Policy* and *RW* performance in relation to the number of agents and papers for Query 4.4 .

Number of papers	**1**	**2**	**3**	**4**	**5**
Number of agents	**3**	**4**	**5**	**6**	**7**
Number of state variables	270	720	1450	2520	3990
X-Policy (time in s)	0.498	1.259	4.889	15.86	134.822
RW (time in s)	0.434	1.354	13.307	48.758	157.352

in the model state (size of the model) and the performance time increases exponentially. However, *X-Policy* still performs better than *RW* in that respect.

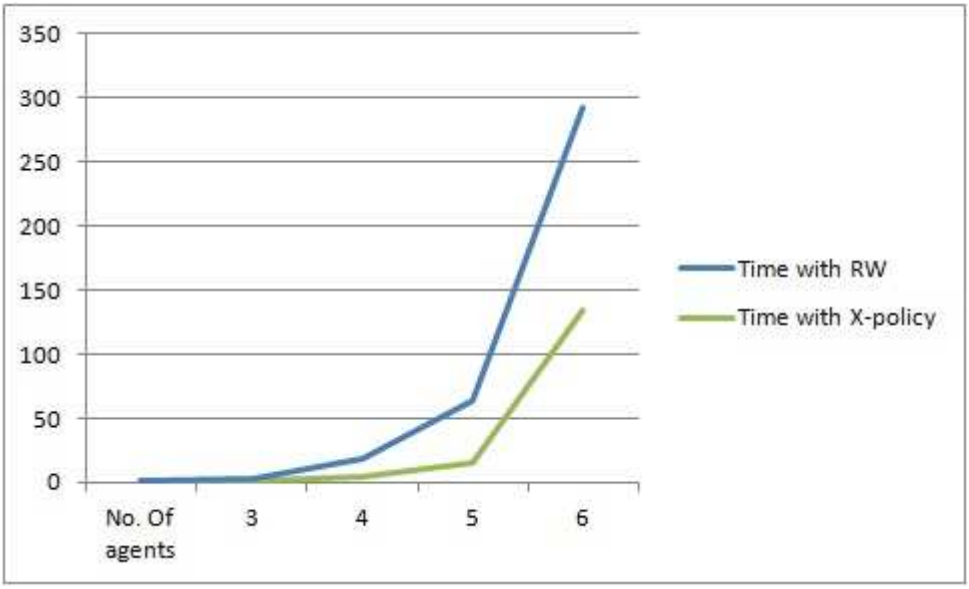

Figure 6.5: Relation between no. of agents and the performance time for *X-Policy* and *RW* .

The computer used for these checks is a PC running Linux Centos (kernel 2.6.18) and Java Runtime Environment 1.6.0_21 with heap size of 2GB on an Intel Core 2 Quad 2.33GHz with 2GB RAM.

The time needed to verify ***EC*** for the properties discussed in 5.4 are as following[1]:

Table 6.3: *X-Policy* performance (all time in sec) when verifying ***EC*** against the properties discussed in 5.4

Property	**Minimum time (s)**	**Maximum time (s)**	**Average time (s)**
Property 1	4.939	7.954	7.2
Property 2	3.883	1.576	1.9
Property 3	0.751	2.840	1.8

6.7 Chapter Summary

In this Chapter we presented the implementation of *X-Policy* tool. We also discussed structure of the tool and the its main components. We followed by explaining the tool usage usage information. The source code structure and overview are discussed in Section 6.3. The semantic checking is discussed in Section 6.4. Section 6.5 explains the notion of computational rounds. We also discussed in detail the tool performance, results and examples. We detail our evaluation of *X-Policy* against *RW* and DyNPAL. Section 6.6.2 includes the performance analysis where we compare the runtime of *X-Policy* against other tools. It also analyse the relation between the size of the model and the runtime needed for *X-Policy* analysis.

[1]Note that due to the space complixy of the algorithm as discussed in Section 4.6 and some implementation issues caused by the garabage collection for the JVM, the tool terminate after a number of runs when the memory used reachs a maximum of 3GB on Linux. The number of runs before the it terminates vary, however, in the worst case scenario at least 10 computational runs where done before it terminated while verifying property 2.

CHAPTER 7

Conclusion and Future Work

This thesis has presented *X-Policy* framework. The motivation for *X-Policy* is to provide a simple modelling language and an automated verification tool to give policy designers and system managers the ability to specify and analyse collaborative system dynamic policies. *X-Policy* framework is designed to model the dynamic execution permissions of large web-based collaborative systems. We use knowledge-based methods to model the multi-agent, centralised and collaborative aspect of these systems.

Access control policies is where three very interesting and fast developing fields meet: software engineering, security engineering and computer science [5, 108]. The requirements and nature of software systems evolve and vary as the focus shift from one technology to another. Often, research in these fields are done in parallel to anticipate or respond to the advances in the other fields. This is certainly true in the case of access control. While the development of the early access control models were taking place as part of the development of the early operating systems, the base of advances to relevant technologies including grid, cloud and service oriented computing has created a new set of problems to be solved.

Security concepts in on-line systems are hard to understand.

On-line systems do not enjoy the physical systems' checks. The ability to have a concert understanding of access control concept of physical systems working makes it easier to understand and analyse these systems. Procedures like keeping the money in a vault or storing patients' record into the clinic archive are easy to understand by the interested parties: patients, doctors and the medical professionals.

The design of access control policies for on-line systems is an an error-prone and inherently difficult process. A general purpose and simple modelling language and an automated analysis tool to model, study and analyse these policies can aid the design process of the access control policy greatly. The modelling language has to be expressive enough to model the concepts of the modelled systems. It also has to be simple enough that users are able to learn and use easily to express concepts in a clear way. This thesis proposed a modelling language and verification tool *X-Policy* that is expressive enough to model real life web-based collaborative systems like EasyChair.

The process of modelling and analysing of access control policies, similar to other security analysis fields [64, 62], depends on the ability to specify the system formally. Model checking-based access control analysis tools like *RW* and *X-Policy* offer us the ability to model the users behaviour and knowledge as well as the working of the system policy. The ability of *X-Policy* to model agents knowledge of the systems state when the user of the system executes compound actions or read variables allows us to distinguish between the legitimate access by the legitimate users and the illegitimate access by attackers. It also allows us to understand the access control policy of the system from the point view of the users. This is often hard as access control policies are developed as part of the system in an ad-hoc fashion.

The ability of the *X-Policy* tool to handle various system sizes in a reasonable time is crucial in our ability to analyse large systems. As Section 6.6.3 show, *X-Policy* tool manages to analyse the access control systems in a reasonably acceptable time compared

to other model checking based tools. However, the state size increases exponentially as we increase the number of variables or agents in the model. The need to represent agents' knowledge is responsible to a great extent for that fast growth of the state. However, the state explosion problem can be improved by using model checking abstraction techniques [26, 27, 68]. More specifically, [63] proposes a method that reduces the size of the model by detecting the effective variables in the model which are relevant to the attacker's strategy. The method first finds the strategy then apply certain checks to ensure that this strategy satisfies the knowledge constraints using the effective variables. This method reduces the number of variables needed and thus reduces the effect of the state explosion problem.

The lack of real-life examples for access control policies formally specified is an issue acknowledged by [11, 15, 16, 121]. This thesis models and analyses the access control policy of EasyChair. We discuss a number of attacks that I discovered as detailed in Chapter 5. These attacks show that simple properties like the author of a paper should not review her own paper can be overlooked while designing and implementing a system and show the need for tools like *X-Policy*. The ability to specify the access control policy in *X-Policy* can simplify this process.

Furthermore, the process of building the specification of an up and running system is labour intensive and requires a lot of patience and experience to get right. While having access to the source code of the system can be intuitive, the actual complexity of the code and the working of the system limits the use of the code in understanding the access control policy. The success and applicability of *X-Policy* to model and verify real-life web-based collaborative systems like EasyChair are demonstrated by modelling and analysing the access control policies for real life web-based collaborative systems like the ones discussed in Chapter 5. *X-Policy* query language allow us to express security properties involving a number of coalitions and nested goals as we discuss in Chapters 3 and 6. Furthermore, our analysis of these systems is aided by the use of *X-Policy*

software implementation. We illustrate the implementation of the tool in Chapter 6. The evaluation of our approach against other tools and approaches is discussed in Chapter 6.

The process of modelling analysing ***EC*** showcase the need to give a considerable attention to the design of the access control policy of the system. The attacks found show that simple implementation flows can occur in the process of developing the system and that can cause some unintended consequences. During the production of this thesis, one of the most interesting reactions to those discovery has been that these attacks are obvious and we may not require the use of automated tool to analyse and discover them. This reaction can be expected for a various of reasons:

- It is often the case for practitioners of a certain field to see the advantages of new methods [115]. Indeed, non-functional requirement like security are often more abstract in their nature and can be hard to explain to project managers

- Security analysis requires a certain way of thinking and expertise. The puzzle like nature of the attack analysis makes the discovery of the attack is where the difficulty lies rather than the nature of the attack itself. This is certainly true in many cases of security attacks [75, 96] and has been attributed to the way we reason about problems based on its representation [117].

- The users of the conference management systems are generally academics and thus they are inclined to preserve their integrity and avoid wrong doing, this argument indeed discounts the human nature. However, the concepts developed in this thesis and tool can be used to model other systems. For example, Moodle [110] is a course management system that is used to provide on-line courses to various audiences. In one particular case, Moodle is used to deliver courses to prisoners [103]. This is indeed one of the cases where ensuring the security of the system is sacrosanct. A security failure of Moodle in this scenario has the potential of not only affecting the

system integrity but the security of the community at large.

Crucially, the approach combines the knowledge acquired by multi-agent coalition using the system with the dynamic behaviour of the compound actions. Our use of model checking algorithm produces attack strategies in when the security property fail. These strategies can help the policy designers understand and fix the policy. *X-Policy*, with its ability to establish the relation between an action and the agent who is executing it, allows us to analyse security properties that require collaboration between a specific set of agents who are allowed to act to achieve an attack on the system. *X-Policy*'s ability to specify read and write actions also allows us to reason about pieces of data being read as part of the attack. Unlike many other tools and languages, the approach allows the expression of reading permissions and reasoning about "under what conditions can an agent execute an action?" rather than on answering the question "under what conditions can an action be executed?". *X-Policy* parameter typing allows us to establish the relation between the agent who can execute an action and the action itself. This is indeed necessary to enable us to define agent coalitions and establish which agent is executing an action. It allows us to detect attacks where we are interested in who can execute a set of actions rather than whether a set of actions can be executed regardless of the agents involved.

The design of *X-Policy* is guided by the use of real life examples and case studies. This proposes a number of modelling conventions for web-based conference management systems which can be applied to other web-based systems. We use EasyChair as our central example for its wide use and importance. We build a model ***EC*** based on our understanding of the policy of EasyChair. The full ***EC*** model is available at [90]. It contains 49 actions and permission statements. This is relatively concise given the size and complexity of EasyChair. The way the system functionality is split into actions is decided by our understanding of how the system is actually designed. The ability to specify multi-assignment actions enables us to maintain the integrity constraint so that,

for example, when a PC-member is deleted, her reviewing assignments are also deleted. Using *X-Policy*, we can reason about the security properties of our model. We presented a case study of three security properties for EasyChair and described the possible attacks on these properties as well as ways the system could be changed to prevent these attacks. We have informed the developer of EasyChair of our findings.

7.1 Summary of Contribution

The main contribution of this thesis is the development of *X-Policy* framework and the analysis of real life large scale systems like EasyChair. More specifically, this thesis provide the following contributions:

- It proposes a novel, simple, parametrised and typed propositional logic-based procedural-like modelling language to model multi-agent dynamic access control policies with the ability to express readability permissions and compound actions.

- It devises a knowledge-based multi-agent model checking algorithm to analyse security properties as reachability queries expressed in *X-Policy* query language.

- It provides the area of verifying dynamic access control policies with the a number of large-scale real-life web-based collaborative systems case-studies.

- It introduces a number of conventions for modelling large scale real life web-based collaborative systems.

- It expresses the access control policy of the modelled systems in *X-Policy* and analysing a number of security properties of these systems.

- It reports the discovery of a number of security flaws in the studied systems and proposing ways to fix these systems.

- It describes the implementation of the software tool for *X-Policy* framework and testing it with a number of case studies.

- It evaluates the *X-Policy* software tool performance and compare it against similar tools.

- It discusses the results of our analysis and the lessons learned from the case study.

While other related approaches try to solve different aspects of the problems of expressing and verifying state-based dynamic access control policies like DYNPAL [12, 14] and *RW* [120, 121, 122], the method in thesis thesis combines the knowledge-based reasoning which is different from DYNPAL and the simple modelling language to express compound actions which *RW* lack. Lo et. al. [74, 87] analyse the security features of Web Submission and Review Software WSAR [46]. They study the security properties like the system password strength and storage, its resistance to SQL injection, forced browsing and browser caching. However, their analysis does not include the access control policy of the system. To the best of our knowledge, this thesis is the first to model and analyse dynamic access control policy for a large web-based collaborative system with atomic actions like EasyChair. This is significant as we discussed in Chapters 2, 5 and 6.

While *X-Policy* provides a simple modelling language with simple constructs, users of *X-Policy* has to decide the appropriate level of abstraction when building the system model. It might be more suitable to abstract some of the actions and variable to let the policy designers target a specific property or issue. While the result of the verification of a model is specific to that certain model and property, the process of building and analysing the model allows for a better understanding and gives the ability to abstract the access control policy in a more effective and uniform way.

X-Policy is best suited for modelling and analysing control-intensive systems like conference management systems. In other applications, such as Facebook, Google+, and

Linkedin, *X-Policy* is able to model relations and actions like forming friendship relations or sharing and commenting on photos. However, the system model of these systems allows other applications to authenticate users credentials to third party services. It also allows these services to access the user's information with the consent of the users. The combination of the authentication process and the dynamic aspect of the access control policy of these services seems interesting. While this is an interesting issue, it falls outside the scope of this thesis.

7.2 Future Work

This thesis should only be seen as another step towards modelling and analysing access control policies in complex real-life systems. This thesis could be extended or build-upon in various directions. Some of the future work can go to address limitations or issues outside the scope and the assumptions of this thesis.

This thesis focuses on web-based collaborative systems is justified by the sustained increase and wide acceptance of centralised highly configurable cloud-based singular profile systems to manage various aspect of out life like the one discussed in Chapter 5. However, other aspects can be considered within the cloud computing model. Cloud computing services attract many users and organisations for its ease of use and operational cost saving benefits. A number of laws exist to regulate data usage and storage one these systems. For example, companies operating in the United Kingdom which collect and store personal data of costumers are required by law to comply with the UK Data Protection Act 1999. Still, cloud computing systems still equally vulnerable as other computing systems as they represent highly prized target as they accumulate massive centralise amount of data. Customers' emails and addresses were stolen as a result of phishing attacks [82]. Our ability to understand these systems is still limited. It is certainly difficult to reason about the trade-off between security and usability in these systems. We still exploring the use

of emerging technologies like trusted computing and TPM devices [106] to ensure certain security properties and establish trust between the system and its users. A wide ranging and diverse approaches, proposals and techniques to analyse and facilitate the design of security requirements of these systems are developed [61, 113, 31, 3, 6, 112]. Many security concerns remain unsolved.

Another aspect of access control policy is temporal constraints[18] and temporal reasoning [59]. For example, we may want to model an access control system for a conference system that uses an actual time as as the deadline for the paper of review submission system. *X-Policy* does not support these constraints at the moment. However, one way to model that is by defining a an agent of the type timer where treat that agent as our clock. We may also need to extend *X-Policy* to define temporal variables that represent a state of the time. This will be interesting issue to attack as an extension to *X-Policy* in the future. At the moment the *X-Policy* query language allows temporal reasoning in the fact that it can define nested query. However, a more expressive temporal reasoning properties can help us analyse these systems where time is critical.

In the case of web-based collaborative systems discussed in this thesis, it will be interesting to explore ways to analyse the side effects of the combination between various systems under different administrators? For example, EasyChair hosts various conference management system simultaneously. The case-studies considered in this thesis analyse the policy across a singular system i.e. a single conference. However, the use of cloud systems allows the users to act in various roles in a number of systems at the same time. A number of questions seems appropriate: can an agent acting in various capacities in a number of conferences gain access to certain information that violate the integrity of one or more of these conferences? Can the EasyChair developers provide guarantees that certain properties regarding information collection and storage are preserved? StatVerif [6] is a process calculus-based tool to verify protocols that affect the global state of the

system which applies to web-based systems and TPM-based systems. There is a need for further research to establish the extend in which our method can be extended to model these properties.

This thesis contributes an analysis algorithm and its implementation as *X-Policy* tool. As we show in Chapter 6 the performance of *X-Policy* is relatively acceptable compared to similar approaches. However, this can be improved. [63] proposed the use of effective variables to reduce the variables considered in the model checking algorithm. Other abstraction techniques [26, 27, 68] can be used to reduce the state variables and increase the size of systems studied. Another possibility is to try a depth-first recursive search in the backward reachability computation. Currently the algorithm for finding strategies uses something like a breadth-first search. However, if we use a depth-first recursive search it may improve the performance of the algorithm in finding strategies. However, a recursive algorithm is more difficult to analyse. Although many challenges remain ahead before we are able to fully understand security requirements for large complex systems like the one discussed in this thesis, the contribution of this work and others facilitate and simplify the process of modelling, studying and analysing these systems

So little done, so much to do.

Alexander Graham Bell [1847–1922]

APPENDIX A

Algo-1

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

```
1  strategies := ∅;
2  states_seen := ∅;
3  put (K_G, skip;) in strategies;
4  repeat until strategies does not change{
5          choose (Y_1, s_1) ∈ strategies;
6          for each act(α⃗) ∈ Actions* {
7                  for each a ∈ C{
8                  PXY := Pre^{∃,a}_{act(α⃗)}(Y_1);
```

```
9                  if ((PXY ≠ ∅) ∧ (PXY ⊈ any set of the pairs in strategies)){
10                         axs_1 := "a : act(α⃗)" + s_1;
11                         if (k_init ∈ PXY)
12                                 output axs_1;
13                         else
14                                 strategies := strategies ∪ {(PXY, axs_1)};
15                         }
16                 }
17         }
18         choose (Y_2, s_2) ∈ strategies;
19         for each variable p(α⃗) ∈ P{
20                 for each a ∈ C{
21                 PRY := Pre^{∃,a}_{p(α⃗)=⊤}(Y_1) ∩ Pre^{∃,a}_{p(α⃗)=⊥}(Y_2);
22                 if ((PRY ≠ ∅) ∧ (PRY ⊈ any set of the pairs in strategies)){
23                         pss :="if (a : p(α⃗)) then s_1 else s_2";
24                         if (k_init ∈ PRY)
25                                 output pss;
26                         else
27                                 strategies := strategies ∪ {(PRY, pss)};
28                         }
29                 }
30         }
31 }
```

APPENDIX B

Step-by-Step derivation of the attacks in Chapter 5

In this Chapter we will provide the step by step derivation the attacks on ***EC*** discussed in Chapter 5. We have discovered these issues while using EasyChair. In each case, we show an attack strategy to achieve an undesirable state by hand.

For the purpose of this analysis we consider the ***EC*** model as defined in Chapter 5. We introduce a number of restrictions to simplify our proof. We restrict our analysis to the system states and we prove a single strategy which is an execution sequence of read and write actions which takes the model from an initial state m_0 to a goal state m_g. A strategy still can be executed by more than one agent where agents collaborate to reach the goal. We show that these strategies work on our model and reach the goal state.

The ***EC*** model defines the sets T, P , $\texttt{Actions}^*$ and permission statements. It also defines, for each type t, a finite set of individuals σ_t. We define $\sigma = \sigma_{t_1} \cup \cdots \cup \sigma_{t_n}$ as the set of all the individuals defined by ***EC***. We assume $\sigma_{t_1} \cap \sigma_{t_2} = \emptyset$ whenever t_1 and t_2 are distinct. If p is a predicate and $\overrightarrow{\alpha}$ is a sequence of individuals of the appropriate type then $p(\overrightarrow{\alpha})$ is a ground atomic formula. State m of the model M is a valuation of the ground atomic formulae. In the rest of this Chapter we identify each state with the set of ground atomic formulae which are true in the state. We also treat the for-loop statements

and calculate the actions effect similarly which we include here for completion.

For loops. We describe the semantics of for-loops in the context of a model $\boldsymbol{EC}$, with $\sigma_t = \{v_1, \ldots, v_k\}$ the set of individuals in $\boldsymbol{EC}$ of the type t. Let $act \in$ Actions. We then transform each for-statement to its equivalent multiple *assignment* statements. For example the following for-statement:

for $(v : t)$ $\{p(\overrightarrow{\gamma_1}, v, \overrightarrow{\gamma_2}) := \bot;\}$

is in the write action $act(\overrightarrow{x})$ where $\overrightarrow{\gamma_1}$ and $\overrightarrow{\gamma_2}$ are subsequences of other parameters. This for-statement is transformed to:

$p(\overrightarrow{\gamma_1}, v_1, \overrightarrow{\gamma_2}) := \bot;$
$\vdots$
$p(\overrightarrow{\gamma_1}, v_k, \overrightarrow{\gamma_2}) := \bot;$

We apply this process repeatedly until we have no for-statement in our action. We call the resulted loop-free action: $act^*(\overrightarrow{x})$.

Effect of Actions. Let $act \in$ Actions and $\overrightarrow{\alpha}$ a sequence of individuals of the appropriate type for act. We define the result of running the instantiated action $act(\overrightarrow{\alpha})$. We first compute $act^*(\overrightarrow{\alpha})$, as above. We then apply the functions: effect$^+$() and effect$^-$() which compute the positive and the negative effect of the instantiated loop-free action $act^*(\overrightarrow{\alpha})$ as following:

$$\text{effect}^+(act^*(\overrightarrow{\alpha})) = \{p(\overrightarrow{\beta}) \text{ — } p(\overrightarrow{\beta}) := \top \text{ occurs in } act^*(\overrightarrow{\alpha})\}$$
$$\text{effect}^-(act^*(\overrightarrow{\alpha})) = \{p(\overrightarrow{\beta}) \text{ — } p(\overrightarrow{\beta}) := \bot \text{ occurs in } act^*(\overrightarrow{\alpha})\}$$

where all the values of $\overrightarrow{\beta}$ are members of σ.

The action effect then will be applied to the model state. Executing a write action will transfer the model from a pre-execution state m_i in which the action is executed at to a post-execution state m_{i+1}. It adds the set of ground atomic formulae which are

updated to true to the state m_i. It also subtracts the set of ground atomic formulae that are updated to false from the state m_i. All the other ground atomic formulae in the state m_i will remain unchanged. Let's say that the model M is in the state m_i when an agent u executes the write action $act(\overrightarrow{\alpha})$, then the model will be transformed from the state m_i to the state m_{i+1} where $m_{i+1} = m_i \backslash \text{effect}^-(act^*(\overrightarrow{\alpha})) \cup \text{effect}^+(act^*(\overrightarrow{\alpha}))$. Note that $\text{effect}^-(act^*(\overrightarrow{\alpha})) \cap \text{effect}^+(act^*(\overrightarrow{\alpha}))=\emptyset$. Therefore, adding and subtracting can be done in any order. Read actions do not change the model state. However, read actions can be part of an attack strategy as we will see in Section 5.4.

We define a number of individuals of types Agent and Paper and use these individuals to define an initial state which we refer to as m_0. For our ***EC*** model, we create the following configuration:

1. The system has five agents: Alice, Bob, Eve, Carol and Marvin. The system has two submitted papers: p1 and p2. We express the configuration as following: $\sigma_{Paper} = \{\text{p1}, \text{p2}\}$ and $\sigma_{Agent} = \{\text{Alice}, \text{Bob}, \text{Carol}, \text{Eve}, \text{Marvin}\}$.

2. Alice is the Chair of PC. Bob and Carol are PC members. Paper p1 is submitted by the author Marvin while p2 is submitted by the author Eve. Reviewers' names are obscured from each other by enabling the anonymous reviewing option. Authors' names are obscured from the PC members and the reviewers. The conference submission is configured in the anonymous submission mode. The list of submissions can be viewed by PC chairs only. Non-chairs do not have access to reviews of papers not assigned to them. In this case, we choose an up-and-running state of ***EC*** to keep our proof to a minimum. However, we can derive the system from an earlier state. We express these settings in the following *X-Policy* configuration:

$m_0 = \{$ Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
PCM-review-editing-en(), View-sub-by-chair-permitted(),
Chair-review-en(), PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled()$\}$

Now that we have defined the initial state m_0, we can analyse the following properties. Each of these properties will start from m_0 and derive the model using a strategy S_i to reach the goal state $m_g^{S_i}$.

B.1 Proof for the attack on property 1

In the following, we use Alice:Add-Reviewer-Assignment(p1,Bob) to denote that the agent Alice executes the action Add-Reviewer-Assignment(p1,Bob). Starting from m_0 we show how the model state evolves through the attack strategy. At each step, we explicitly list, when appropriate, the list of ground atomic formulae that has to be absent or present to execute the following action from the model state:

Alice:Add-Reviewer-Assignment(p1,Bob);

requires the presence of Chair(Alice), PCmember(Bob), Review-assig-enabled().

requires the absence of Conf-of-interest(p1,Bob).

$m_1^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob)}

Alice:Add-Reviewer-Assignment(p1,Carol);

requires the presence of Chair(Alice), PCmember(Carol), Review-assig-enabled().

requires the absence of Conf-of-interest(p1,Carol).

$m_2^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol)}

Bob:Request-Reviewing(p1,Bob,Eve);

requires the presence of PCmember(Bob), Reviewer(p1,Bob), PCM-review-menu-en().

$m_3^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol), Requested-subrev(p1,Bob,Eve)}

Carol:Request-Reviewing(p1,Carol,Eve);

requires the presence of PCmember(Carol), Reviewer(p1,Carol), PCM-review-menu-en().

$m_4^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol), Requested-subrev(p1,Bob,Eve), Requested-subrev(p1,Carol,Eve)}

Eve:Accept-Reviewing-Request(p1,Bob,Eve);

requires the presence of Requested-subrev(p1,Bob,Eve).

$m_5^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol), Requested-subrev(p1,Bob,Eve), Requested-subrev(p1,Carol,Eve), Decided-subrev(p1,Bob,Eve), Subreviewer(p1,Bob,Eve)}

Eve:Accept-Reviewing-Request(p1,Carol,Eve);

requires the presence of Requested-subrev(p1,Carol,Eve).

$m_6^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol), Requested-subrev(p1,Bob,Eve), Requested-subrev(p1,Carol,Eve), Decided-subrev(p1,Bob,Eve), Subreviewer(p1,Bob,Eve), Decided-subrev(p1,Carol,Eve), Subreviewer(p1,Carol,Eve)}

Bob:Add-Review(p1,Bob,Eve);

requires the presence of PCmember(Bob), Reviewer(p1,Bob), PCM-review-menu-en(), PCM-review-editing-en().

$m_7^{S_1}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(), Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol), Requested-subrev(p1,Bob,Eve), Requested-subrev(p1,Carol,Eve), Decided-subrev(p1,Bob,Eve), Subreviewer(p1,Bob,Eve), Decided-subrev(p1,Carol,Eve), Subreviewer(p1,Carol,Eve), Submitted-review(p1,Bob,Eve)}

Carol:Add-Review(p1,Carol,Eve);

requires the presence of PCmember(Carol), Reviewer(p1,Carol),PCM-review-menu-en(), PCM-review-editing-en().

$m_8^{S_1}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Sub-anonymous(),
Author(p2,Eve), PCM-review-editing-en(), View-sub-by-chair-permitted(),
Chair-review-en(), PCM-review-menu-en(), Review-assig-enabled(), Reviewer(p1,Bob), Reviewer(p1,Carol),
Requested-subrev(p1,Bob,Eve), Requested-subrev(p1,Carol,Eve),
Decided-subrev(p1,Bob,Eve), Subreviewer(p1,Bob,Eve), Decided-subrev(p1,Carol,Eve),
Subreviewer(p1,Carol,Eve), Submitted-review(p1,Bob,Eve), Submitted-review(p1,Carol,Eve)}

In this case the model state m_o has evolved during the scenario to the goal state $m_8^{S_1}$. The ground atomic formulae Submitted-review(p1,Bob,Eve) and Submitted-review(p1,Carol,Eve) are in $m_8^{S_1}$. This means that Eve has managed to write two reviews for the same paper and get them submitted to the system. Similarly, Eve could have written all the reviews of that particular paper. Consequently, EasyChair fails the property as a single reviewer can determine the outcome of a paper.

B.2 Proof for the attack on property 2

We now show how the model state evolves through the attack strategy. At each step, we explicitly list, when appropriate, the list of ground atomic formulae that has to be absent or present to execute the following action from the model state:

Alice:Add-Reviewer-Assignment(p2,Bob);
requires the presence of Chair(Alice), PCmember(Bob), Review-assig-enabled().
requires the absence of Conf-of-interest(p2,Bob).

$m_1^{S_2}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(),
PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p2,Bob)}

Bob:Request-Reviewing(p2,Bob,Eve);
requires the presence of PCmember(Bob), Reviewer(p2,Bob), PCM-review-menu-en().

$m_2^{S_2}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(),

PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p2,Bob),
Requested-subrev(p2,Bob,Eve)}

Eve:Accept-Reviewing-Request(p2,Bob,Eve);

requires the presence of PCmember(Bob), Reviewer(p2,Bob), PCM-review-menu-en().

$m_3^{S_2}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(),
PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p2,Bob),
Decided-subrev(p2,Bob,Eve), Subreviewer(p2,Bob,Eve)}

Bob:Add-Review(p2,Bob,Eve);

requires the presence of PCmember(Bob), Reviewer(p2,Bob), PCM-review-editing-en(),
PCM-review-menu-en().

$m_4^{S_2}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(),
PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p2,Bob),
Decided-subrev(p2,Bob,Eve), Subreviewer(p2,Bob,Eve), Submitted-review(p2,Bob,Eve)}

In this case the model has evolved to the goal state where ground atomic formula Submitted-review(p2,Bob,Eve) is in $m_4^{S_2}$. This means that Eve has managed to submit a review for her own paper p2. EasyChair fails this property as it allows a paper's reviewers to submit a review written by the paper's author herself.

B.3 Proof for the attack on property 3

We show how the model state evolves through the attack strategy:

Alice:Add-Reviewer-Assignment(p1,Bob);

requires the presence of Chair(Alice), PCmember(Bob)Review-assig-enabled().
requires the absence of Conf-of-interest(p1,Bob)

$m_1^{S_3}$ = {Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve),
Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(),
PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p1,Bob)}

Bob:Add-Review(p1,Bob,Bob);

requires the presence of PCmember(Bob), Reviewer(p1,Bob), PCM-review-menu-en(), PCM-review-editing-en().

$m_2^{S_3}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve), Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p1,Bob), Submitted-review(p1,Bob,Bob)}

Alice:Show-Review(p1,Bob,Bob);

requires the presence of Chair(Alice), Chair-review-en().

$m_3^{S_3}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve), Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p1,Bob), Submitted-review(p1,Bob,Bob)}

Alice:Add-Review(p1,Carol,Carol);

requires the presence of Chair(Alice), PCmember(Bob), Reviewer(p1,Carol), PCM-review-menu-en(), PCM-review-editing-en().

$m_4^{S_3}$ = { Chair(Alice), PCmember(Bob), PCmember(Carol), Author(p1,Marvin), Author(p2,Eve), Conf-of-interest(p1,Alice), PCM-review-editing-en(), View-sub-by-chair-permitted(), Chair-review-en(), PCM-review-menu-en(), Sub-anonymous(), Review-assig-enabled(), Reviewer(p1,Bob), Submitted-review(p1,Bob,Bob), Submitted-review(p1,Carol,Carol)}

We can see that $m_4^{S_3}$ is the goal state $m_g^{S_3}$ as chair Alice has managed to submit a review to the paper p1 as if she were the PC member Carol.

APPENDIX C

Syntax of the X-policy language

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

⟨Model⟩ ::= ⟨System⟩ [⟨RunStatement⟩] [⟨Specification⟩]

⟨System⟩ ::= "AccessControlSystem" ⟨SystemName⟩ ⟨Body⟩ "End"

⟨Body⟩ ::= [⟨TypeDefSection⟩] ⟨PredicateDefSection⟩ ⟨Rules⟩

⟨TypeDefSection⟩ ::= "Type" ⟨TypeName⟩ ("," ⟨TypeName⟩)* ";"

⟨PredicateDefSection⟩ ::= "Predicate"⟨PredicateDef⟩ ("," ⟨PredicateDef⟩)* ";"

⟨PredicateDef⟩ ::= ⟨PredicateName⟩(("("")")|("(" ⟨ParameterName⟩ ":" ⟨ClassName⟩
("," ⟨ParameterName⟩ ":"⟨ClassName⟩)* ")"))["!"]

⟨Rules⟩ ::= ⟨ReadRules⟩ ⟨ActionRules⟩

⟨ReadRules⟩ ::= ⟨ReadRule⟩ (⟨ReadRule⟩)*

⟨ActionRules⟩ ::= ⟨ActionRule⟩ (⟨ActionRule⟩)*

⟨ReadRule⟩ ::= ⟨AccessPattern⟩ "{" [⟨ReadStatement⟩] "}"

⟨AccessPattern⟩ ::= ⟨PredicateName⟩ "("⟨FormalParameter⟩ ("," ⟨FormalParameter⟩)* ")"

⟨ReadStatement⟩ ::= "read" ":" ⟨Formula⟩ ";"

⟨ActionRule⟩ ::= ⟨ActionName⟩ ⟩((“(”“)”)|(“(” ⟨ParameterName⟩ “:” ⟨ClassName⟩

(“,” ⟨ParameterName⟩ “:”⟨TypeName⟩)* “)”))

“{” [⟨WriteSubroutine⟩] “}”

“{” [⟨ActionStatement⟩]“}”

⟨WriteSubroutine⟩::=[⟨SingleAssignment⟩]|[⟨MultiAssignment⟩]

(⟨MultiAssignment⟩)*(⟨SingleAssignment⟩)*

⟨SingleAssignment⟩::=⟨SinglePredicate⟩ “:=”(“true”|“false”)“;”

⟨MultiAssignment⟩::=“for(” ⟨LocalVariable⟩ “:” ⟨TypeName⟩ “)”

“{”⟨SingleAssignment⟩ (⟨SingleAssignment⟩)*“}”

⟨ActionStatement⟩ ::= ⟨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⟩ ::= [“dist”§] ⟨QuantifiedVariable⟩ (“,” ⟨QuantifiedVariable⟩)*

“:” ⟨TypeName⟩

⟨ModelName⟩ ::= ⟨Id⟩

⟨TypeName⟩† ::= ⟨Id⟩

⟨PredicateName⟩ ::= ⟨Id⟩

⟨ParameterName⟩‡ ::= ⟨Id⟩

⟨FormalParameter⟩ ::= ⟨Id⟩

⟨QuantifiedVariable⟩ ::= ⟨Id⟩

⟨LocalVariable⟩::= ⟨Id⟩

⟨implies⟩ ::= “implies” | “→”

⟨or⟩ ::= “or” | “|”

⟨and⟩ ::= “and” | “&”

⟨negation⟩ ::= “∼”

⟨Id⟩ ::= ⟨Letter⟩ (⟨Letter⟩ | ⟨Digit⟩ | “_” | “-”)*

† ⟨TypeName⟩ must start with an upper case letter.

‡ ⟨ParameterName⟩ must start with a lower case letter.

§ The keyword “dist” can only be used on quantified variables defined in ⟨CheckStatement⟩

The precedence is: “=” > “∼” > “&” > “|” > “→”

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

⟨NumberClassPair⟩ ::= ⟨Integer⟩ ⟨ClassName⟩

⟨Specification⟩ ::= ⟨CheckStatement⟩

⟨CheckStatement⟩ ::= “check” “{” ⟨QuantifiedVariablesList⟩ “||”
[⟨Conditions⟩ “→”] ⟨Coalition⟩ “:” ⟨Goal⟩ “}”

⟨QuantifiedVariablesList⟩ ::= “E”|“A” ⟨QuantifiedVariablesDef⟩
(“,” [“E”|“A”] ⟨QuantifiedVariablesDef⟩)*

⟨Conditions⟩ ::= ⟨Condition⟩ (⟨and⟩ ⟨Condition⟩)*

⟨Condition⟩ ::= ⟨PositiveCondition⟩ | ⟨NegativeCondition⟩

⟨PositiveCondition⟩ ::= ⟨PredicateName⟩ “(”⟨FormalParameter⟩
(“,” ⟨FormalParameter⟩)*“)” “*” | “!” | “*!”

⟨NegativeCondition⟩ ::= ⟨negation⟩ ⟨PredicateName⟩ "("⟨FormalParameter⟩
(","⟨FormalParameter⟩)*")" "!" | "*!"

⟨Goal⟩ ::= ["("] ⟨OrGoal⟩ [⟨SubGoal⟩] [")"]

⟨SubGoal⟩ ::= "AND" ⟨Coalition⟩ ":" "("⟨OrGoal⟩ (⟨SubGoal⟩)* ")"

⟨OrGoal⟩ ::= ⟨AndGoal⟩ (⟨or⟩ ⟨AndGoal⟩)*

⟨AndGoal⟩ ::= ⟨AtomicGoal⟩ (⟨and⟩ ⟨AtomicGoal⟩)*

⟨AtomicGoal⟩ ::= ⟨ReadingGoal⟩ | ⟨MakingGoal⟩ | "("⟨Goal⟩")"

⟨ReadingGoal⟩ ::= "["⟨GoalExpression⟩"]"

⟨MakingGoal⟩ ::= "{"⟨GoalExpression⟩"}"

⟨GoalExpression⟩ ::= ⟨OrGoalExpression⟩ (⟨implies⟩ ⟨OrGoalExpression⟩)*

⟨OrGoalExpression⟩ ::= ⟨AndGoalExpression⟩ (⟨or⟩ ⟨AndGoalExpression⟩)*

⟨AndGoalExpression⟩ ::= ⟨BasicGoalExpression⟩ (⟨and⟩ ⟨BasicGoalExpression⟩)*

⟨BasicGoalExpression⟩ ::= ⟨AGoalPredicate⟩ | ⟨negative⟩ ⟨BasicGoalExpression⟩
| "("⟨GoalExpression⟩")"

⟨AGoalPredicate⟩ ::= ⟨SinglePredicate⟩

⟨Coalition⟩ ::= "{" ⟨Id⟩ ("," ⟨Id⟩)*"}"

APPENDIX D

Models used to evaluate *X-Policy* against similar tools.

In this Section, we illustrate the models used to evaluate *X-Policy* against similar tools in Section 6.6.2. In Chapter 6 we used commonly used examples of access control systems as first introduced by [121] and later used for evaluating DyNPAL in [12]. In the following we list the *X-Policy* script for each system and we detail the Queries used as specified in *X-Policy*.

D.1 Employee information system (EIS)

Employee information system (EIS) describes the access control policy regarding the process of allocating bonuses to employees by managers and directors. It also allows union representatives to advocate for other employee. The *X-Policy* script for the employee information system is given in Listing D.1.

```
AccessControlSystem EmployeeInformationSystem
        Type Bonus;
        Predicate bonus(employee: Agent, bonus: Bonus),
```

```
        manager(employee: Agent),
        director(employee: Agent),
        advocate(appointer: Agent, appointee: Agent);

        bonus(a,b){
        read: (user=a or director(user))
        or (manager(user) and ~manager(a) and ~director(a))
        or (advocate(a,user)); }
        manager(a){ read: true;    }
        director(a){ read: true;   }
        advocate(a1,a2){read:true;  }

        Action Setbonus(a:Agent,b: Bonus)
        {bonus(a,b):=true;}
        {(manager(user) and ~manager(a) and ~director(a))
                or director(user);}
        Action UnSetbonus(a:Agent,b: Bonus)
        {bonus(a,b):=false;}
        {(manager(user) and ~manager(a) and ~director(a))
                or director(user);}
        Action Setmanager(a:Agent)
        {manager(a):=true;}
        {(director(user) and (user=a and manager(a) and ~director(a)));}
        Action UnSetmanager(a:Agent)
        {manager(a):=false;}
        {(director(user) and (user=a and manager(a) and ~director(a)));}
        Action Setadvocate(a1:Agent,a2:Agent)
        {advocate(a1,a2):=true;}
        {user=a1 or (user=a2 and advocate(a1,a2));}
End
```

Listing D.1: *X-Policy* script for the employee information system.

D.2 Student information system (SIS)

Student information system (SIS) which describes the access control policies for the process of allocating marks to students for a given course. Each course has a lecturer and a number of demonstrators who can be students as long as they are in a higher year than the students taking that course which we detail in Listings D.2.

```
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; }
        mark(a){ read: user=a; }

        Action SetMark(a:Agent)
        {mark(a):=true;}
        {lecturer(user) | demonstrator_of(user,a);}
        Action AssignDem_of(d: Agent, s: Agent)
        {demonstrator_of(d,s):=true;}
        { (lecturer(user) & higher(d,s))
                or (demonstrator_of(d,s) & user=d);       }
        Action De-assignDem_of(d: Agent, s: Agent)
        {demonstrator_of(d,s):=false;}
        {(lecturer(user) & higher(d,s))
                or (demonstrator_of(d,s) & user=d);}
End
```

Listing D.2: *X-Policy* script for the student information system.

D.3 Conference review system (CRS)

Conference review system (CRS) describes the access control policy regarding the process of submitting and reviewing papers in a conference which is encoded in the *X-Policy* script in Listings D.3.

```
AccessControlSystem Conference
        Type  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;     }
        reviewer(p, a){          read : pcmember(user)&~author(p, user);
            }
        subreviewer(p, a, b){    read : (pcmember(user)&~author(p,user))
            or user=b or user=a;     }
        submittedreview(p, a){   read : pcmember(user)&~author(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);
        }
        Action EditeReview(p: Paper, a: Agent)
        {review(p, a):=true;}
        {user=a&((E b: Agent [subreviewer(p, b, user)])|reviewer(p,a)) &
             ~submittedreview(p, user);}

        Action SubmitReview(p: Paper, a: Agent)
        {submittedreview(p, a):=true;}
        {(user=a)&((E b: Agent [subreviewer(p, b, user)])
                or reviewer(p, user)) and ~submittedreview(p, user);}

        Action AssignSubreviewer(p: Paper, a: Agent, b: Agent)
        {subreviewer(p, a,b):=true;}
        { (reviewer(p,a)&~author(p,b)&user=a&~(E d: Agent [subreviewer(p
            ,a,d) or subreviewer(p,d,b)]));}
        Action DeAssignSubreviewer(p: Paper, a: Agent,  b: Agent)
        {subreviewer(p, a,b):=false;}
        {(subreviewer(p,a,b)&~submittedreview(p,b)&user=b);}
        Action AssignReviewer(p: Paper, a: Agent)
        {reviewer(p, a):=true;}
        {(chair(user)&pcmember(a)&~author(p,a));}
        Action DeAssignReviewer(p: Paper, a: Agent)
        {reviewer(p, a):=false;}
        {(((pcmember(user)&user=a&reviewer(p,user))&~(E b:Agent[
            subreviewer(p,user,b)])));}
        Action AssignPCmember(a:Agent){pcmember(a):=true;}{chair(user);}
        Action DeassignPCmember(a:Agent){pcmember(a):=false;}{(pcmember(
            a) and a=user);}
End
```

Listing D.3: *X-Policy* script for conference review system.

APPENDIX E

EC full model in *X-Policy*

In the following, we include the ***EC*** model discussed in Chapter 5 encoded in *X-Policy*

```
AccessControlSystem EC
Type Paper;
Predicate  Author(p:Paper,a:Agent),
Chair-review-menu-enabled(),
Chair-status-menu-enabled(),
Chair(a:Agent),
Conflict-of-interest(p:Paper,a:Agent),
Decided-subreviewing(p:Paper,a:Agent,b:Agent),
PCmember-access-reviews-enabled(),
PCmember-review-editing-enabled(),
PCmember-review-menu-enabled(),
PCmember-status-menu-enabled(),
Accessed-subreviewing(p:Paper,a:Agent, b:Agent),
PCmember(a:Agent),
Requested-subreviewing(p:Paper,a:Agent,b:Agent),
Review-Assignment-enabled(),
Reviewer(p:Paper,a:Agent),
Show-reviewer-name() ,
Submission-anonymous(),
Submissions-open(),
Submitted-review(p:Paper,a:Agent, b:Agent) ,
Subreviewer(p:Paper,a:Agent,b:Agent),
```

```
Updated-review(p:Paper,a:Agent, b:Agent),
View-submission-by-chair-permitted(),
View-submission-by-pcmember-permitted() ,
View-submission-title-permitted(),
View-submission-txt-permitted();
Conflict-of-interest(p,a)
{read:(Chair(user))  or (a=user and PCmember(user)
and Reviewer(p,user)
and Conflict-of-interest(p,a));}
PCmember(a){
 read:Chair(user);
 }
 Submitted-review(p,a,b)
{
read:((Chair(user) and Chair-review-menu-enabled())
  or(PCmember(user) and Reviewer(p,user)
and PCmember-review-menu-enabled()
and E c: Agent [Submitted-review(p,user,c)])
  or(PCmember(user) and ~ Reviewer(p,user)
and PCmember-review-menu-enabled()
and View-submission-title-permitted()
and PCmember-access-reviews-enabled()
and ~ Conflict-of-interest(p,a)));}
 Author(p,a){
 read: (Chair(user) or(~ Submission-anonymous() and PCmember(user)
and(Reviewer(p,user) or View-submission-title-permitted()) )
or Accessed-subreviewing(p,a,user) );}
Chair(a) {
read :  Chair(user);}
PCmember-review-editing-enabled(){read: Chair(user) or PCmember(user);}
Action ShowSubmissionTitleandText()
{
View-submission-title-permitted():=true;
View-submission-txt-permitted():=true;
}
{  Chair(user) ;
}
Action ShowSubmissionNone() // do not show PCs submission text and title.
{
View-submission-title-permitted():=false;
View-submission-txt-permitted():=false;
}
{
```

```
 Chair(user) ;
}

Action ShowSubmissionTitle() // show PCs submission title only.
{
View-submission-title-permitted():=true;
View-submission-txt-permitted():=false;
}
{ Chair(user) ;}
Action ShowSubmissions2Chair() // show all submissions to chair only.
{
View-submission-by-chair-permitted():=true;
View-submission-by-pcmember-permitted() :=false;
}
{Chair(user) ;}

Action ShowSubmissions2All() // show all submissions to chair + PC-member.
{
View-submission-by-chair-permitted():=true;
View-submission-by-pcmember-permitted() :=true;
}
{Chair(user);}

Action ShowSubmissions2None() // show all submissions to no-one.
{
View-submission-by-chair-permitted():=false;
View-submission-by-pcmember-permitted() :=false;
}
{ Chair(user);}

Action EnableStatusM4Chair() // enable status menu for chair.
{
Chair-status-menu-enabled():=true;
PCmember-status-menu-enabled():=false;
}
{ Chair(user);}

Action EnableStatusM4All() // enable status menu for PC-chair+ PC members
{
Chair-status-menu-enabled():=true;
PCmember-status-menu-enabled():=true;
}
{ Chair(user);}
```

```
Action EnableStatusM4None()// enable status menu for No-one.
{
Chair-status-menu-enabled():=false;
PCmember-status-menu-enabled():=false;
}
{ Chair(user);}

Action EnableReview4Chair()  // enable review menu for chair only.
{
Chair-review-menu-enabled():=true;
PCmember-review-menu-enabled():=false;
}
{ Chair(user);}

Action EnableReviewMenu4All()  // enable review menu for chair+PCs.
{
Chair-review-menu-enabled():=true;
PCmember-review-menu-enabled():=true;
}
{ Chair(user);}

Action EnableReviewMenu4None()  // enable review menu for no-one.
{
Chair-review-menu-enabled():=false;
PCmember-review-menu-enabled():=false;
}
{ Chair(user);}

Action ShowReviewerName()  // show other PC members other reviewers name.
{
Show-reviewer-name():=true;
}
{ Chair(user);}

Action HideReviewerName()  // hide reviewers name from other PC members.
{
Show-reviewer-name():=false;
}
{ Chair(user);}

Action EnableReviewEditting()  // enable non-chairs to add or modify reviews.
{
```

```
PCmember-review-editing-enabled():=true;
}
{ Chair(user);}

Action DisableReviewEditting()  // disable non-chairs to add or modify reviews.
{
PCmember-review-editing-enabled():=false;
}
{ Chair(user);}



Action EnableOtherReviewsAccess()  // enable PCs to view other papers reviews.
{
PCmember-access-reviews-enabled():=true;
}
{ Chair(user);}

Action DisableOtherReviewsAccess()  // disable PCs to view other papers reviews
{
PCmember-access-reviews-enabled():=false;
}
{ Chair(user);}

Action EnableAnonymousSubmission()  // Hide the submissions authors.
{
Submission-anonymous():=true;
}
{ Chair(user);}

Action DisableAnonymousSubmission()  // Show the submissions authors.
{
Submission-anonymous():=false;
}
{ Chair(user);}

Action OpenSubmission()  // open the system to accept other submissions.
{
Submissions-open():=true;
}
{ Chair(user);}

Action CloseSubmission()  // close the submissions system.
```

```
{
Submissions-open():=false;
}
{ Chair(user);}

Action EnableAssignment()  // enable chair to(de-)assign papers to PCs.
{
Review-Assignment-enabled():=true;
}
{ Chair(user);}

Action DisableAssignment()  // disable chair to(de-) assign papers to PCs.
{
Review-Assignment-enabled():=false;
}
{ Chair(user);}

Action AddPCmember(a:Agent)
{
PCmember(a):=true;
}
{ Chair(user);}
Action AddChair(a:Agent)
{
Chair(a):=true;
}
{ Chair(user);}
Action PromotePCmemberToPCchair(a:Agent)
{
PCmember(a) :=false ;
Chair(a):=true;
}
{ Chair(user);}

Action DemotePCchairToPCmember(a:Agent)
{
PCmember(a) :=true ;
Chair(a):=false;
}
{ Chair(user);}

Action DeletePCmember(a:Agent)
{
```

```
PCmember(a) :=false;
Chair(a):=false ;
for(p:Paper){ Reviewer(p,a):=false;}
}
{ Chair(user);}

Action AddConflictofInterest(p:Paper,a:Agent)
{
Conflict-of-interest(p,a):=true;
Reviewer(p,a):=false;
}

 {( (Chair(user) and a=user) or(a=user and PCmember(a)
 and(Reviewer(p,a) or View-submission-title-permitted())
  and ~ Conflict-of-interest(p,a)));}

Action RemoveConflictofInterest(p:Paper,a:Agent)
{
Conflict-of-interest(p,a):=false;
}
{ Chair(a) and Conflict-of-interest(p,a);}

Action DeleteReviewerAssignment(p:Paper,a:Agent)
{
Reviewer(p,a):=false;
for(b:Agent) {Subreviewer(p,a,b):=false;}
}
{Chair(user) and Reviewer(p,a);}

Action DeleteReview(p:Paper,a:Agent)
{
Submitted-review(p,a):=false;
}
{ Chair(user) and Submitted-review(p,a);}

Action AddReview(p:Paper,a:Agent,b:Agent)
{
Submitted-review(p,a,b) :=true;
}
{((Chair(user)
and Chair-review-menu-enabled())
((a=user or Chair(user)) and
((PCmember(user) and Reviewer(p,user)
```

```
and PCmember-review-menu-enabled()
and PCmember-review-editing-enabled())
or( PCmember(user) and ~ Reviewer(p,user)
and PCmember-review-menu-enabled()
and View-submission-title-permitted()
and PCmember-access-reviews-enabled()
and ~ Conflict-of-interest(p,a)))))
;}
Action Updatereview(p:Paper,a:Agent,b:Agent)
{
Updated-review(p,a,b):=true;
}
{(Submitted-review(p,a,b) and
((Chair(user)
and Chair-review-menu-enabled())
((a=user or Chair(user)) and
((PCmember(user) and Reviewer(p,user)
and PCmember-review-menu-enabled()
and PCmember-review-editing-enabled())
or( PCmember(user) and ~ Reviewer(p,user)
and PCmember-review-menu-enabled()
and View-submission-title-permitted()
and PCmember-access-reviews-enabled()
and ~ Conflict-of-interest(p,a))))));}
Action RequestReviewing(p:Paper,a:Agent,b:Agent)
{
Requested-subreviewing(p,a,b):=true;
}
{((Chair-review-menu-enabled()
and Chair(user) and E c:Agent [ Author(p,c)])
  or(PCmember(user)
and Reviewer(p,user)
and PCmember-review-menu-enabled())
or(PCmember(user)
and ~ Reviewer(p,user)
and PCmember-review-menu-enabled()
and View-submission-title-permitted()
and PCmember-access-reviews-enabled()
and ~ Conflict-of-interest(p,a)));}
Action AcceptReviewingRequest(p:Paper,a:Agent,b:Agent)
{
Decided-subreviewing(p,a,b):=true;
Subreviewer(p,a,b):=true;
```

```
}
{(Requested-subreviewing(p,a,b) and   E c:Agent [ Author(p,c)]
and(~ Decided-subreviewing(p,a,user) or user = a));}

Action RejectReviewingRequest(p:Paper,a:Agent,b:Agent)
{
Decided-subreviewing(p,a,b):=true;
Subreviewer(p,a,b):=false;
}{ (Requested-subreviewing(p,a,b) and  E c:Agent [ Author(p,c)]
and(~ Decided-subreviewing(p,a,user) or user = a));}
Action AccessReviewingRequest(p:Paper,a:Agent,b:Agent)
{
Accessed-subreviewing(p,a,b):=true;
}
{ (Requested-subreviewing(p,a,b)
and  E c:Agent [ Author(p,c)]
and(~ Decided-subreviewing(p,a,user) or user = a));}
Action SubmitPaper(p:Paper,a:Agent)
{
Author(p,a):=true;
}
{ user=a;}

Action AddAuthor(p:Paper,a:Agent)
{
Author(p,a):=true;
}
{ Author(p,user);}

Action DeleteAuthor(p:Paper,a:Agent)
{
Author(p,a):=false;
}
{Author(p,user) and ~(a=user);}

Action WithdrowSubmission(p:Paper)
{
for(a:Agent){ Author(p,a):=false;}
}
{Author(p,user);}


Action AddReviewerAssignment(p:Paper,a:Agent)
```

```
{
Reviewer(p,a):=true;
}
{Chair(user) and(PCmember(a) or a=user)
and E c:Agent [ Author(p,c)]  and ~ Conflict-of-interest(p,a);}
End
```

The Queries tried on ***EC*** are:

- Property 1: A single subreviewer should not be able to determine the outcome of a paper reviewing process by writing two reviews of the same paper.

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and  PCmember(Bob)! and  PCmember(Carol)!
        and Author(p1,Marvin)! and Author(p2,Eve)!
        and PCmember-review-editing-enabled()!
        and View-submission-by-chair-permitted()!
        and      Chair-review-menu-enabled()!
        and PCmember-review-menu-enabled()!
        and Submission-anonymous()!
        and  Review-Assignment-enabled()!
            -> {Alice,Carol,Bob}:( {Submitted-review(p1,Bob,Eve)}
                THEN {Alice,Carol,Bob}:{Submitted-review(p1,
                    Carol, Eve)})}
```

- Property 2: A paper author should not review her own paper.

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and  PCmember(Bob)! and  PCmember(Carol)!
        and  Author(p1,Marvin)! and Author(p2,Eve)!
        and  PCmember-review-editing-enabled()!
        and  View-submission-by-chair-permitted()!
```

```
and  Chair-review-menu-enabled()!
and  PCmember-review-menu-enabled()!
and   Submission-anonymous()!
and  Review-Assignment-enabled()!
        -> {Alice,Carol,Bob,}: {Submitted-review(p2,Bob,Eve)}
           }
```

- Property 3: Users should be accountable for their actions.

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and  PCmember(Bob)! and  PCmember(Carol)!
        and  Author(p1,Marvin)! and Author(p2,Eve)!
        and   PCmember-review-editing-enabled()!
        and  View-submission-by-chair-permitted()!
        and  Chair-review-menu-enabled()!
        and  PCmember-review-menu-enabled()!
        and  Submission-anonymous()!
        and  Review-Assignment-enabled()!
           -> {Alice,Bob}: {Submitted-review(p1,Carol,Carol)} }
```

- Property 4: Chair cannot assign a PC-member to review a paper if she has a conflict of interest with it.

```
run for 2 Paper, 5 Agent
check {E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)*! and  PCmember(Bob)! and  PCmember(Carol)*!
        and  Author(p1,Marvin)! and Author(p2,Eve)!
        and  PCmember-review-editing-enabled()!
        and  Conflict-of-interest(p1, Carol)*!,
        and  View-submission-by-chair-permitted()!
        and  Chair-review-menu-enabled()!
```

```
and  PCmember-review-menu-enabled()!
and  Submission-anonymous()!
and  Review-Assignment-enabled()!
        -> {Alice}: {Reviewer(p1,Carol)} }
```

- Property 5: a reviewer cannot read another reviewer's review of a paper assigned to her before she submits her review of the paper if the Chair choose to restrict the viewing of the paper submissions to the relevant PC-members..

```
run for 2 Paper, 5 Agent
check {E dist p1:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
        Chair(Alice)! and  PCmember(Bob)! and  PCmember(Carol)!
        and  Author(p1,Marvin)!
        and  PCmember-review-editing-enabled()!
        and  View-submission-by-chair-permitted()!
        and  Chair-review-menu-enabled()!
        and  PCmember-review-menu-enabled()!
        and  Submission-anonymous()!
        and  Reviewer(p1,Bob)
        and  Submitted-review(p1,Carol,Carol)
        and  Review-Assignment-enabled()!
            ->{Bob}:([Submitted-review(p1,Carol,Carol)] AND {Bob,
                Alice}:({Submitted-review(p1,Bob,Bob)}))}
```

List of References

[1] M. Abadi. Logic in access control. In *In Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, volume 15, pages pages 228–233, Ottawa, Canada, June 2003. IEEE Computer Society Press. One citation in section 2.2.1.

[2] M. Abadi, M. Burrows, Lampson B., and Plotkin G. A calculus for access control in distributed systems. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 15, pages 706–734. ACM Press, Sep 1993. 2 citations in sections 2.2.1 and 2.2.1.

[3] Sarah Al-Azzani and Rami Bahsoon. Using implied scenarios in security testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 15–21, New York, NY, USA, 2010. ACM. One citation in section 7.2.

[4] R. Alur, L. deAlfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. *Mocha User Manual*. The manual and the model checker can be obtained from `http://www.eecs.berkeley.edu/~mocha`. One citation in section 2.3.

[5] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. One citation in section 7.

[6] Myrto Arapinis, Eike Ritter, and Mark Ryan. Statverif: Verification of stateful processes. In *Proceedings of the 24th IEEE Computer Security Foundations symposium*, pages 33–47. CSF '11, IEEE Computer Society Press, 2011. One citation in section 7.2.

[7] Tuomas Aura and Carl Ellison. Privacy and accountability in certificate systems. Research Report A61, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, April 2000. One citation in section 2.2.2.

[8] Steve Barker, Marek J. Sergot, and Duminda Wijesekera. Status-based access control. *ACM Trans. Inf. Syst. Secur.*, 12(1):1:1–1:47, October 2008. One citation in section 2.1.1.

[9] L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. Consumable credentials in logic-based access control, 2006. One citation in section 2.2.1.

[10] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and semantics of a decentralized authorization language. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 3–15, 2007. 4 citations in sections 2.2.2, 2.2.3.3, 2.4, and 5.2.3.

[11] Moritz Y. Becker. *Cassandra: flexible trust management and its application to electronic health records*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. 3 citations in sections 2.2.2, 2.2.3.2, and 7.

[12] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. *Computer Security Foundations Symposium, IEEE*, 0:203–217, 2009. 6 citations in sections 2.3.3, 2.4, 6.6.2, 6.6.3, 7.1, and D.

[13] Moritz Y. Becker, Cedric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language,. Technical report, Microsoft Research, September 2006. 2 citations in sections 2.2.2 and 2.2.3.3.

[14] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In *European Symposium on Research in Computer Security*, 2007. 3 citations in sections 2.3.3, 2.4, and 7.1.

[15] Moritz Y. Becker and Peter Sewell. Cassandra: distributed access control policies with tunable expressiveness. 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), 2004. 4 citations in sections 2.2.2, 2.2.3.2, 2.4, and 7.

[16] Moritz Y. Becker and Peter Sewell. Cassandra: flexible trust management, applied to electronic health records (.pdf). 17th IEEE Computer Security Foundations Workshop (CSFW), 2004. 3 citations in sections 2.2.2, 2.2.3.2, and 7.

[17] D. Bell and L. La Padula. Secure computer systems: unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corporation, July 1975. One citation in section 2.1.1.

[18] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001. One citation in section 7.2.

[19] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999. One citation in section 2.2.2.

[20] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society, May 1996. 2 citations in sections 2.2.2 and 2.2.2.1.

[21] Piero Bonatti. Datalog for security, privacy and trust. In *Datalog Reloaded*, Lecture Notes in Computer Science, pages 21–36. Springer Berlin, 2011. One citation in section 2.2.2.

[22] A J Bonner and M Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994. One citation in section 2.3.3.

[23] Anthony J Bonner and Michael Kifer. *Transaction Logic Programming*, pages 257–279. Number CSRI-270 in Logic Programming. The MIT Press, 1993. One citation in section 2.3.3.

[24] Christian Caryl. Why Wikileaks changes everything. *The New York Review of Books*, 58(1):9–13, January 2011. One citation in section 1.1.

[25] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, and Marco Roveri. *NuSMV 2.4 user manual*, 2005. This document and the model checker can be obtained from `http://nusmv.irst.itc.it`. One citation in section 2.3.

[26] 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. 3 citations in sections 2.3.1, 7, and 7.2.

[27] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. 2 citations in sections 7 and 7.2.

[28] Haim Cohen, John Whaley, and J∅rn Lind-Nielsen. BuDDy: Binary decision diagram package, Jun 2004. The package can be obtained from `http://sourceforge.net/projects/buddy`. One citation in section 6.1.

[29] Tom Copeland. *Generating Parsers with JavaCC, Second Edition.* Centennial Books, Alexandria, VA, 2009. One citation in section 6.1.

[30] Robert Craven, Jorge Lobo, Alessandra Russo, Emil Lupu, Jiefei Ma, Arosha Bandara, Morris Sloman, and Seraphin Calo. A formal framework for policy analysis. Technical report, Department of Computing , Imperial College London, 2008. available at `http://www.doc.ic.ac.uk/~rac101/ffpa/`. One citation in section 2.1.1.

[31] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. A formal analysis of authentication in the TPM. In *Workshop on Issues in the Theory of Security*, pages 111–125, 2010. One citation in section 7.2.

[32] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002. 2 citations in sections 2.2.2 and 2.4.

[33] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software Practice and Experience*, 33:397–421, 2003. One citation in section 2.1.

[34] Blair Dillaway. A unified approach to trust, delegation, and authorization in large-scale grids,. Technical report, Microsoft Corporation, September 2006. 2 citations in sections 2.2.2 and 2.2.3.3.

[35] Alistair Donaldson and Phil Walker. Information governanceŮa view from the nhs, realizing security into the electronic health record. *International Journal of Medical Informatics , Elsevier Science*, Volume 73, Issue 3:281–284, 2004. One citation in section 2.2.3.2.

[36] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme, September 1997. One citation in section 2.3.2.

[37] 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. 3 citations in sections 2.3, 2.3.2, and 2.4.

[38] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007. One citation in section 2.2.1.

[39] Michael Gelfond and Jorge Lobo. Authorization and obligation policies in dynamic systems. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 22–36, Berlin, Heidelberg, 2008. Springer-Verlag. One citation in section 2.1.1.

[40] Michael Matthew Greenberg, Casey Marks, Leo Alexander Meyerovich, and Michael Carl Tschantz. The soundness and completeness of margrave with respect to a subset of xacml. Technical Report CS-05-05, Computer Science Department, Brown University, April 2005. One citation in section 2.3.2.

[41] T Gruber. Collective knowledge systems: Where the Social Web meets the Semantic Web. *Web Semantics Science Services and Agents on the World Wide Web*, 6(1):4–13, 2007. One citation in section 1.1.

[42] Dimitar P. Guelev, Mark D. Ryan, and Pierre-Yves Schobbens. Model-checking access control policies. In *7th Information Security Conference (ISC'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004. 2 citations in sections 2.3 and 4.3.2.

[43] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *In Proc. 27th ACM Symp. on Principles of Programming Languages POPL*, pages 316–329. ACM Press, 2000. One citation in section 2.2.2.

[44] Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foun-*

dations Symposium, volume 0, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society. One citation in section 2.4.

[45] Yuri Gurevich and Itay Neeman. DKAL 2 : A simplified and improved authorization language. Technical report, Microsoft Research - Cambridge, 2009. 2 citations in sections 2.4 and 5.2.3.

[46] Shai Halevi. Sourceforge.net: Web submission and review software. Available online at `http://sourceforge.net/projects/websubrev`. One citation in section 7.1.

[47] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, Pacific Grove, California, 2003. One citation in section 2.1.1.

[48] Zahera Harb. Arab revolutions and the social media effect. *MC Journal*, 14(2), 2011. One citation in section 1.1.

[49] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976. One citation in section 2.1.1.

[50] Michael V. Hayden. *National Information Systems Security Glossary*. National Security Telecommunications and Information Systems Security Committee, Sep 2000. One citation in section 2.1.1.

[51] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi-terminal binary decision diagrams to represent and analyse continuous-time markov chains, 1999. One citation in section 2.3.2.

[52] J Hoffmann and B Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001. One citation in section 2.3.3.

[53] Daniel Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 256–290. ACM Press, Apr 2002. One citation in section 2.3.

[54] Daniel Jackson. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. Software Design Group, MIT Lab for Computer Science, Feb 2002. This document and the tool can be obtained from http://alloy.mit.edu/. 2 citations in sections 2.3 and 3.2.3.

[55] Daniel Jackson. *Alloy 3.0 Reference Manual*, May 2004. This document and the tool can be obtained from http://alloy.mit.edu/. One citation in section 2.3.

[56] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the Alloy constraint analyser. In *22nd international conference on Software engineering*, pages 730–733. ACM Press, 2000. One citation in section 2.3.

[57] JavaCC project group. JavaCC project, Aug 2011. Information about JavaCC and jjTree can be found at https://javacc.java.net/. One citation in section 6.1.

[58] T. Jim. SD3: a trust management system with certified evaluation. In *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pages 106 –115, 2001. One citation in section 2.2.2.

[59] J B D Joshi, E Bertino, and A Ghafoor. An analysis of expressiveness and design issues for the generalized temporal role-based access control model. *Ieee Transactions On Dependable And Secure Computing*, 2(2):157–175, 2005. One citation in section 7.2.

[60] Cecilia Kang and William Wan. Google hack gives way to diplomatic, high-tech tensions. *The Washington Post*, 3rd of June 2011. One citation in section 1.1.

[61] R Kazman, M Klein, M Barbacci, T Longstaff, H Lipson, and J Carriere. The architecture tradeoff analysis method. *Proceedings Fourth IEEE International Conference on Engineering of Complex Computer Systems Cat No98EX193*, 98(c):68–78, 1998. One citation in section 7.2.

[62] Arthur M Keller, David Mertz, Joseph Lorenzo Hall, and Arnold Urken. Privacy issues in an electronic voting machine. *Proceedings of the 2004 ACM workshop on Privacy in the electronic society WPES 04*, page 33, 2004. One citation in section 7.

[63] Masoud Koleini and Mark Ryan. A knowledge-based verification method for dynamic access control policies. In *Proceedings of 13th International Conference on Formal Engineering Methods (ICFEM 2011).*, 2011. 2 citations in sections 7 and 7.2.

[64] Steve Kremer, Mark Ryan, and Ben Smyth. *Election verifiability in electronic voting protocols*, volume 6345. Springer-Verlag, 2010. One citation in section 7.

[65] B. Lampson. Computer security in the real world. Presented at the Annual Computer Security Applications Conference, 2000, 2001. One citation in section 2.2.1.

[66] Butler Lampson. Protection and access control in operating systems. In *Operating Systems*, pages 309–326. Infotech State of the Art Report, 1972. One citation in section 2.1.1.

[67] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992. 3 citations in sections (document), 2.2.1, and 2.3.

[68] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for ctl model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, San Jose, CA, Nov 1996. 2 citations in sections 7 and 7.2.

[69] N. Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, New York, Sep 2000. 2 citations in sections 2.2.2 and 2.2.3.1.

[70] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*, January 2003. 3 citations in sections 2.2.1, 2.2.2, and 2.2.3.1.

[71] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland*, May 2002. 3 citations in sections 2.2.2, 2.2.3.1, and 2.4.

[72] Ninghui Li, John C. Mitchell, William H. Winsborough, Kent E. Seamons, Michael Halcrow, and Jared Jacobson. Rtml: A role-based trust-management markup lan-

guage. Technical report, Purdue University, 2004. CERIAS TR 2004-03. 2 citations in sections 2.2.2 and 2.2.3.1.

[73] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *ACM Conference on Computer and Communications Security*, pages 156–165, 2001. 2 citations in sections 2.2.2 and 2.2.3.1.

[74] Swee-Won Lo, Raphael C.-W. Phan, and Bok-Min Goi. On the Security of a Popular Web Submission and Review Software (WSaR) for Cryptology Conferences. In *WISA '07: the 8th International Workshop on Information Security Applications*, Lecture Notes in Computer Science. Springer, 2007. One citation in section 7.1.

[75] G Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995. One citation in section 7.

[76] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998. One citation in section 2.2.2.

[77] D. McDermott and J. Doyle. Nonmonotonic logic 1. *Artificial Intelligence*, 13:41–72, 1980. 2 citations in sections 2.2.2 and 2.4.

[78] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Mar 1999. One citation in section 2.3.

[79] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993. One citation in section 2.3.

[80] Stephan Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, pages 3–38. Springer-Verlag, 2001. One citation in section 4.6.

[81] Toby Miller and Pal Ahluwalia. Wikileaks looks up. *Social Identities*, 17(2):167–167, 2011. One citation in section 1.1.

[82] Miranda Mowbray and Siani Pearson. A client-based privacy manager for cloud computing. *Proceedings of the Fourth International ICST Conference on COMmu-*

nication System softWAre and middlewaRE COMSWARE 09, page 1, 2009. One citation in section 7.2.

[83] Qun Ni, Elisa Bertino, and Jorge Lobo. D-algebra for composing access control policy decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 298–309, New York, NY, USA, 2009. ACM. One citation in section 2.1.1.

[84] Alexander Nicoll. WikiLeaks: the price of sharing data. *Strategic Comments*, 17(1):1–3, 2011. One citation in section 1.1.

[85] Jaehong Park and Ravi Sandhu. The uconabc usage control model. *ACM Trans Inf Syst Secur*, 7(1):128–174, 2004. One citation in section 2.1.1.

[86] Ben Parr. Biggest Military Leak in History: WikiLeaks Releases 390,000 Iraq War Documents, October 2010. One citation in section 1.1.

[87] Raphael C.-W. Phan and Huo-Chong Ling. On the insecurity of the Microsoft Research Conference Management Tool (MSRCMT) system. In *CITA 2005: Proceedings of 4th International Conference on IT in Asia*, pages 75–79, 2005. One citation in section 7.1.

[88] Hasan Qunoo. X-Policy: Knowledge-based verification tool for dynamic access control policies. Technical Report CSR-11-09, University of Birmingham, School of Computer Science, December 2011. No citations.

[89] Hasan Qunoo, Masoud Koleini, and Mark Ryan. Towards modelling and verifying dynamic access control policies for web-based collaborative systems. W3C Workshop on Access Control Application Scenarios, November 2009. No citations.

[90] Hasan Qunoo and Mark Ryan. EC model in X-policy. online at `http://www.cs.bham.ac.uk/~hxq/X-policy/`, Dec 2009. 3 citations in sections 5.2.1, 5.3, and 7.

[91] Hasan Qunoo and Mark Ryan. Modelling dynamic access control policies for web-based collaborative systems. In Sara Foresti and Sushil Jajodia, editors, *DBSec*, volume 6166 of *Lecture Notes in Computer Science*, pages 295–302. Springer, 2010. One citation in section 1.1.

[92] Hasan Qunoo and Mark Ryan. Modelling dynamic access control policies for web-based collaborative systems - long version. Technical report, School of Computer Science, University of Birmingham, April 2010. One citation in section 5.4.

[93] Hasan Qunoo and Mark Ryan. Modelling dynamic access control policies for web-based collaborative systems. Technical Report CSR-11-08, University of Birmingham, School of Computer Science, August 2011. No citations.

[94] Peter Z. Revesz. *Introduction to Constraint Database*. Springer, 2002. One citation in section 2.2.3.2.

[95] Mark D Ryan. Cloud computing privacy concerns on our doorstep. *Communications of the ACM*, 54(1):36, 2011. One citation in section 1.1.

[96] Peter Ryan and Steve Schneider. An attack on a recursive authentication protocol. a cautionary tale. *Information Processing Letters*, 65(1):7–10, 1998. One citation in section 7.

[97] Ravi Sandhu. Separation of duties in computerised information systems. In *IFIP WG11.3 Workshop on Database Security*, Sep 1990. One citation in section 2.3.

[98] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, Feb. 1999. One citation in section 2.3.

[99] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996. 2 citations in sections 2.1.1 and 2.1.1.

[100] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 1994. One citation in section 2.1.1.

[101] Andreas Schaad and Jonathan Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT'02*, Monterey, California, USA, Jun 2002. One citation in section 2.3.

[102] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a European bank: a case study and discussion. In *SACMAT'01*, Chantilly, Virginia, USA, May 2001. ACM Press. One citation in section 2.3.

[103] Niall Sclater. Enhancing moodle to meet the needs of 200,000 distance learners. In *Silesian Moodle Moot*. Technical University of Ostrava, Celadnzech Republic, 30-31 Oct 2008. One citation in section 7.

[104] Geotechnical Software Services. Java programming style guidelines. The document was obtained at `http://geosoft.no/development/javastyle.html`, Apr. 2007. One citation in section 6.1.

[105] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. A role-based access control policy verification framework for real-time systems. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 13–20, Washington, DC, USA, 2005. IEEE Computer Society. One citation in section 2.3.

[106] S.W. Smith. *Trusted computing platforms: design and applications*. Springer, 2005. One citation in section 7.2.

[107] Fabio Somenzi. CUDD: Colorado University decision diagram package 2.4.1, May 2005. 2 citations in sections 2.3.2 and 6.1.

[108] Ian Sommerville. *Software Engineering*. Pearson Education, 9 edition, 2010. One citation in section 7.

[109] Nikhil Swamy, Juan Chen, and Ravi Chugh. *Enforcing Stateful Authorization and Information Flow Policies in Fine*, volume 6012, pages 529–549. Springer, 2010. One citation in section 2.2.1.

[110] the Moodle Trust. Moodle open source course management system (cms). `http://www.moodle.org`. One citation in section 7.

[111] David Toman. Memoing evaluation for constraint extensions of datalog. *Constraints*, 2(3-4):337–359, 1997. One citation in section 2.2.3.2.

[112] S Uchitel, R Chatley, J Kramer, and J Magee. Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. In *Joint European Conference on Theory and Practice of Software (ETAPS 2003)*, pages 597–601. Springer-Verlag Berlin, 2003. One citation in section 7.2.

[113] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004. One citation in section 7.2.

[114] J. Ullman. *Database and Knowledge-Base Systems*. Computer Science Press, 1989. One citation in section 2.2.2.

[115] Eric Von Hippel. Lead users: A source of novel product concepts. *Management Science*, 32(7):791–805, 1986. One citation in section 7.

[116] Andrei Voronkov. EasyChair stats. `http://www.voronkov.com/easychair.cgi/`. 2 citations in sections 5 and 5.1.

[117] P C Wason and Diana Shapiro. Natural and contrived experience in a reasoning problem. *The Quarterly Journal Of Experimental Psychology*, 23(1):63–71, 1971. One citation in section 7.

[118] John Whaley. JavaBDD: Java BDD implementation, 2004. Information about this implementation can be found at `http://javabdd.sourceforge.net/`. One citation in section 6.1.

[119] Nan Zhang. Verification of access control systems using Mocha. Master's thesis, School of Computer Science, University of Birmingham, 2002. One citation in section 2.3.

[120] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems in XACML. In *2004 ACM Workshop on Formal Methods in Security Engineering*, pages 56–65, Washington DC, USA, Oct 2004. ACM Press. 3 citations in sections 2.3, 2.4, and 7.1.

[121] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 2005. 14 citations in sections 1.1, 1.2, 2, 2.3, 3.2.3, 3.2.4, 4.1, 4.3.2, 6.6.2, 6.6.3, 6.1, 7, 7.1, and D.

[122] Nan Zhang, Mark D. Ryan, and Dimitar P. Guelev. Evaluating access control policies through model-checking. In *8th Information Security Conference (ISC'05)*, Singapore, Sep 2005. Springer-Verlag. 3 citations in sections 2.3, 2.3.1.1, and 7.1.