VERIFICATION OF TEMPORAL-EPISTEMIC PROPERTIES OF ACCESS CONTROL SYSTEMS

by

MASOUD KOLEINI

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
December 2011

Abstract

Verification of access control systems against vulnerabilities has always been a challenging problem in the world of computer security. The complication of security policies in large-scale multi-agent systems increases the possible existence of vulnerabilities as a result of mistakes in policy definition.

This thesis explores automated methods in order to verify temporal and epistemic properties of access control systems. While temporal property verification can reveal a considerable number of security holes, verification of epistemic properties in multi-agent systems enable us to infer about agents' knowledge in the system and hence, to detect unauthorized information flow.

This thesis first presents a framework for knowledge-based verification of dynamic access control policies. This framework models a coalition-based system, which evaluates if a property or a goal can be achieved by a coalition of agents restricted by a set of permissions defined in the policy. Knowledge is restricted to the information that agents can acquire by reading system information in order to increase time and memory efficiency. The framework has its own model-checking method and is implemented in Java and released as an open source tool named *PoliVer*.

In order to detect information leakage as a result of reasoning, the second part of this thesis presents a complimentary technique that evaluates access control policies over temporal-epistemic properties where the knowledge is gained by reasoning. We will demonstrate several case studies for a subset of properties that deal with reasoning about knowledge. To increase the efficiency, we develop an automated abstraction refinement technique for evaluating temporal-epistemic properties.

For the last part of the thesis, we develop a sound and complete algorithm in order to identify information leakage in DATALOG-based trust management systems.

ACKNOWLEDGEMENTS

I owe my deepest gratitude to my supervisor, Mark Ryan. This thesis would not have been possible without his guidance, careful evaluation of my work, and patience in facing my mistakes. During the three years of my studies, he has taught me many technical and non-technical things, pointed out my weaknesses and tried to help me in overcoming them.

I would like to thank my supervisor at Microsoft, Moritz Becker. He kindly provided me with the opportunity of an internship at Microsoft Research Cambridge and the chance to work with him. During my internship, he generously taught me research, programming and writing skills which influenced my work since.

I am also grateful to Dimitar Guelev, who introduced me to interpreted systems. His guidance and discussions resulted in an important part of this thesis. I should also thank my supervisor, Mark Ryan, for inviting Dimitar to Birmingham to help us with our research.

Special thanks go to my mother, Esmat Davani, who brought me up and did everything for our education and improvement in all difficult circumstances of our life. I also thank my brother Mehrshad Koleini for all his kindness and support for the family. I would like to thank my wife, Arezoo Hosseini, for her full support and patience during my studies.

As well as all the above, I wish to thank Eike Ritter for his great collaboration and also being a member of my thesis group, James Margetson for his full support whenever I had problem in programming in F#, Samin Ishtiaq and other nice and intelligent Microsoft people for making my time at Microsoft Research Cambridge an exciting experience, my thesis group member Tom Chotia for his helpful recommendations, Peter Hancox who always openly accepted me for asking questions, and all the staff of the Computer Science department.

Microsoft®
Research

This research is sponsored by Microsoft Research.

Statement of collaboration

The research in chapter 4 and 5 is my own work in collaboration with Mark Ryan (both chapters) and Eike Ritter (chapter 5). The developement of the open source tool (PoliVer) and the implementation and experiments in chapter 5 are my own work. Chapter 6 presents the research in Microsoft Research Cambridge by Moritz Becker and me. The main idea is Moritz Becker's and the research was driven and technically written by him. I participated mainly in the implementation and experimental results and partly in the technical proofs.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Research motivation

Social networks like Facebook and LinkedIn, cloud computing networks like Salesforce and Google docs, conference paper review systems like Easychair and HotCRP are examples of applications that huge numbers of users deal with every day. In such systems, a group of agents interact with each other to access resources and services. Such multi-agent collaborative systems are getting more and more complex which raises the possibility of there being vulnerabilities in their information access rules.

All the above systems have a built-in access control system with a set of rules, named *access control policy*. Policy designers have a human readable form of the access policy for the principals in the system which should be enforced. Further developments of the system cause the access control policy to become more complicated and as a consequence, it may not comply with the organization information security requirements. For such complex systems, reasoning about the correctness of access control policy by hand is not feasible. Automated verification is a solution which enables policy designers to verify their policies against required properties. For instance, in Google docs, we need to verify "if Alice shares a document with Bob, it is not possible for Bob to share it with Charlie unless Alice agrees", or in HotCRP, "if Bob is not chair, it is not possible for him to promote himself to be a reviewer of a paper submitted to the conference". If such properties do not hold, it can imply a security hole in the system that needs to be investigated and fixed by policy designers.

One of the most challenging aspects of verifying access control systems is *knowledge*, which is the information that an agent or group of agents gain about the system. *Information leakage* is the knowledge that is acquired by some unauthorized principals. In general, finding information leakage in the systems is not straightforward. This is because not all the information is gained by direct access, but some is gained by reasoning. Successful hackers are in general talented in reasoning. They use the information gathered by social engineering together with the ones gained by interacting with the system in order to find a way to penetrate the system.

An important question that arises is: Is it possible to verify the knowledge that a principal can acquire in a system which is regulated by a set of access control rules? To answer this question, we first divide access control policies into two categories: *dynamic* or *state-based*, and *stateless*. In dynamic policies [94, 11, 42, 78], the permissions for an agent depend on the state of the system. As a consequence, permissions for an agent can be changed by the actions of other agents. In such policies, the knowledge also may change when the state changes. In stateless policies [13, 70, 48], access decision does not change the state of the system.

We first look at the category of dynamic policies, which is the main contribution of this thesis. While the majority of the research is focused on verification of temporal properties, formal reasoning for information leakage and anonymity is not well automated by the state of the art tools [88]. One of the frameworks for the verification of temporal-epistemic properties over dynamic policies is proposed by Zhang et al. [95, 94] which is implemented as a tool named AcPeg. Although their framework models a memoryful system (perfect recall) by building the system around knowledge states, it suffers from several problems. We will discuss the problems later in the related work section, but in the context of knowledge verification, they only consider the knowledge gained by reading system information. Their framework is not able to verify the information leaked as a result of reasoning. As a positive point, modelling knowledge by the information gained by reading system information reduces the complexity and improves the efficiency. On the other hand, this approach is unable to detect some specific but important information leakage vulnerabilities in the policies.

In the category of stateless policies, the first research that adequately formalized the information flow in trust management frameworks was performed by Becker [12]. He proposes an algorithm which is able to find if some private information can be leaked to an unauthorized principal in a Datalog-based policy through sending legitimate credentials, called *probes*, to the system. His work is well formalized and the soundness of the algorithm is formally proved. Becker's approach has two weaknesses: first, the algorithm that investigates if some information is detectable in a policy is sound but not complete, and second, the algorithm is difficult to automate. Therefore, there is still the requirement of finding a sound and complete algorithm which is also easier to implement.

1.2 Knowledge-based verification in dynamic policies

This research was motivated at the beginning by RW, the formal verification framework developed by Zhang, Ryan and Guelev [94, 95] at the University of Birmingham. RW (Read and Write) is built around the states that store the knowledge of the agents or in the other words, *knowledge states* and uses model-checking techniques for property verification. The knowledge in RW is the accumulation of the knowledge the agents in a coalition gain by reading system variables. Our main idea was that building the system around knowledge states is not essential when the knowledge is the result of reading system information. In the case of knowledge by readability and when perfect recall is required, reading a variable can be introduced into the policy as an action together with some extra variables that support memory of reading. In the original RW framework, introducing memory into the policy is not possible as we require actions to update multiple variables at the same time. In RW, each write action can only update one variable at a time. Our idea resulted in designing and implementing a new tool, which we named *PoliVer*[1].

Verification of knowledge by readability is simpler and more efficient than the knowledge gained by reasoning. While a high percentage of vulnerabilities can be detected by RW, PoliVer, DynPAL and other policy verification tools, there still exists some information leakage vulnerabilities that the state of the art tools are not able to detect. Let us demonstrate it with an example.

Example 1.1. Assume a conference paper review system in which all the PC members have access to the number of papers assigned to each reviewer. Further assume that each PC member can see the list of the papers assigned to the reviewers which does not contain the papers that he is the author of.

An important security requirement in a conference paper review system is that no author should be able to find out who is the reviewer of his or her paper. This property does not hold in the above system. Assume that Alice is a PC member and also the author of a paper which is submitted to the conference and Bob is allocated as the reviewer of her paper. By the policy, Alice knows how many papers Bob is allocated to review. When Alice checks the list of the papers assigned to Bob, she finds that the number of the papers she can see in the list is less than total number of papers assigned to Bob. Therefore, Alice can reason that Bob is the reviewer of her paper, which violates the security requirement in the system. Such information leakage vulnerabilities can not be detected by the tools that model knowledge by readability of information.

In addition and in this research, we develop a method that enables us to verify the

[1] PoliVer is available at `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/`

knowledge expressed by the modal logic KT45^n [58] using *Interpreted systems framework* [45, 46]. We also use abstraction and refinement techniques in order to overcome time and memory limitations when verifying temporal-epistemic properties.

Model-checking and abstraction This research uses model-checking as the formal verification method. The *Interpreted systems framework* allows us to reason about knowledge in multi-agent systems, and will be used as our framework later in this research. Model-checking temporal-epistemic properties often becomes intractable when the number of variables and therefore the state space grows. For large state space systems, abstraction techniques can be adopted in order to simplify the model. Abstraction in multi-agent systems has received few attention in recent years [33, 96, 44]. As another important point, automated refinement methods for epistemic properties is not as developed as those for temporal properties [30, 91]. Therefore, design and development of automated abstraction and refinement is of considerable importance when realistic large systems come into account. This research also contributes to designing an automated abstraction refinement in order to optimize the verification of temporal-epistemic properties over access control systems.

1.3 Information leakage in static policies

The detectability and opacity of information in trust management systems in not fully investigated yet. In distributed systems, the trust management framework is subject to some attacks called *probing attacks* which were first introduced by Becker [12]. Gurevich and Neeman [50] demonstrated a similar attack on SecPAL [13], which is a Datalog-based policy language. In probing attacks, an adversary can infer information about the system by submitting a series of *probes*, which are access requests together with conditional credentials. To demonstrate the problem, let's have a look to the following example [12, 50]:

Example 1.2. Imagine a service policy, which allows principals to park their car according to some terms and conditions. The service policy also contains some confidential facts like if a principal is a secret agent written in SecPAL [13]:

`Service` says x `can park` if x `consents to parking rules`
`Service` says x can say x `consents to parking rules`

In the above policy, says denotes the intension or digital signature of an agent over the assertions, and `can say` denotes the delegation of authority. The query ⟨`Service` says

`Alice can park`⟩ succeeds if the assertions `Service` says `Alice consents to parking rules` evaluates to true, which can result from the second assertion union with the assertion `Alice` says `Alice consents to parking rules`. Now Alice submits two self-issued credentials together with the request for parking permission to the service:

1. `Alice` says `Alice consents to parking rules` if `Bob is a secret agent`
2. `Alice` says `Service` can say `Bob is a secret agent`

If the above assertions together with the query ⟨`Service` says `Alice can park`⟩ succeeds, then we need to find out if the fact `Service` says `Bob is a secret agent` is crucial for the succession of the query. Therefore, Alice needs another step in order to complete the attack. She submits only the first credential together with the query. In the case of denial of the permission, Alice can infer that Bob is a secret agent.

For complicated policies in Datalog-based trust management systems, the problem of decidability of opacity (the negation of detectability) is open. The third part of this research solves this problem by proposing a sound and complete algorithm that has the power to determine if a property is opaque in a given policy.

1.4 Our solution

For the category of dynamic policies and in this thesis, we cover verifying interesting temporal-epistemic properties over dynamic access control policies. We divide our approach into two categories and compare the outcomes:

Knowledge gained by reading (Chapter 4):

1. We develop a model-checking framework that deals with knowledge by readability. We will show that even in the case of memoryful knowledge, it is not practically efficient to verify such properties in knowledge state. We prove this argument by comparing the runtime and memory usage of our algorithm with RW, which works on knowledge space. To have a memoryful approach, the memory of reading system variables can be introduced into the policy and therefore system states instead of incorporating knowledge states for each agent.
2. Based on our framework, we implement the tool *PoliVer*. PoliVer keeps some useful features of RW (guessing strategies) as we found them to be important in verification of properties over access control policies. The policy language is expressive and the query language is reach enough to handle nested queries. We apply a post

processing algorithm in order to verify the knowledge of agents over the information that agents require in order to achieve the goal. Finally, we compare the results with its predecessor AcPeg. PoliVer is implemented in Java and performs symbolic model-checking using the binary decision diagram library BuDDy [71].

Knowledge gained by reasoning: (Chapter 5)

1. We introduce the second framework as the complimentary which is now able to verify epistemic properties that deal with reasoning. The new framework is less time and memory efficient, but is able to detect information leakage vulnerabilities in policies which are not possible to be detected by the state of art verification tools. We use interpreted systems as our basic framework.

2. Our framework in general uses a larger number of state space and higher verification time compared to our approach in the first category (knowledge by readability). In order to make the verification method competitive and more practical for large systems, we design an automated abstraction and refinement method for temporal-epistemic safety properties which dramatically reduces the time and memory consumption. Our abstraction and refinement method is applicable when verifying safety properties.

Datalog-based policies: (chapter 6)

Regarding to the category of DATALOG-based authorization systems, Becker first introduced a method that detects information leakage by verifying if a property is detectable. As we discussed before, the algorithm is not complete and is difficult to implement. In this research, we verify the detectability by checking if a property is *not opaque*. We propose the first sound and complete algorithm which is also easy to implement. As verifying knowledge in DATALOG-based policies grows exponentially when the number of available probes increases, similar to the dynamic systems, we need to apply optimization. Hence, we develop several methods that reduce the state space search and then compare the results with the ones in the absence of optimization.

1.5 Research contribution

Our research focuses on finding automated methods in order to check if required properties hold in an access control policy. Our research consists of three parts:

First, we investigate knowledge-based verification of properties over access control policies when the knowledge comes from reading system information. In this research, we implement and release a model-checking tool called PoliVer, with the following properties:

- The tool handles co-operation of agents in a collaborative environment, together with interaction of the rules and multi-step actions.

- It provides a user friendly syntax that covers action rules, which are able to update a group of system variables at one step, and read permission rules, that define the conditions in which system variables are allowed to be read by the agents.

- The query language is flexible and supports nested goals with possible different coalitions active for each goal.

- The model-checking algorithm finds the propositions that the coalition requires their value in order to proceed through the goal. A complementary algorithm checks if the agent is able to find the values of those propositions or needs to take the risk of guessing the values.

- Various case studies demonstrate the experimental results and enhancements over the previous knowledge-based approach (RW framework [94, 95]).

Second, we develop a complimentary verification method based on interpreted systems with the ability of reasoning about the knowledge of the agents in access control systems. In this part of the research, we use a model-checker for multi-agent systems called MCMAS[1] [74, 72] as our model-checking engine, and build our framework on top of it. Our approach has the following properties:

- The policy language is similar to the one for PoliVer, which supports action rules and read permission rules. The query language is the standard CTLK (CTL logic with knowledge modality K).

- Time and memory usage is reduced by implementing a fully automated abstraction and refinement algorithm over temporal-epistemic properties. Although the optimization is mainly available for safety properties in ACTLK (CTLK containing only universal path quantifier), we also provide an interactive refinement for some security properties that do not fall into the category of ACTLK.

- In the absence of abstraction, all properties in CTLK logic can be verified over the policy. Without optimization, the verification may not be practical for medium to large policies.

- The verification tool is implemented in F# programming language. Our case studies compare the results of our new approach with PoliVer and RW.

[1] MCMAS is available at `http://www-lai.doc.ic.ac.uk/mcmas/`

Thirdly, we carefully investigate information leakage vulnerability in Datalog-based trust management systems. The outcome of the result is a tool written in F# functional programming language, which accepts an input policy and available probes, and determines whether a property is opaque in the policy or not. This research has the following properties:

- A formal definition for probing attacks over trust management systems is provided, which is handled in verifying opacity in Datalog-based policies. Datalog-based policies are vastly used in various trust management frameworks.
- Our algorithm is proved to be sound, complete and terminating: If it proves the opacity of a property when the property is not detectable (soundness), and if it fails to prove the opacity, then the property is provably detectable. The proof procedure always terminates assuming that the number of available probes for the adversary is finite.
- Several optimization mechanisms are provided. Our experimental results show that they dramatically reduce verification time in many practical scenarios.
- Several realistic case studies demonstrate the effectiveness of the algorithm. The verification times for different optimization methods are also calculated and compared with non-optimized algorithm.

1.6 Structure of the thesis

The reminder of the thesis is structured as follows: Chapter 2 covers the related work and background of access control policy verification. This chapter also looks at the abstraction refinement techniques in model-checking as it will be used in our research. Chapter 3 is the preliminary chapter, which provides the required materials like policy language and some definitions for the rest of the thesis. In Chapter 4, we present a verification algorithm that investigates temporal-epistemic properties over an access control system described by a policy and when the knowledge is gained by the readability of information. We demonstrate the performance of the implemented tool in the experimental results. Chapter 5 contains a complimentary framework that is able to verify epistemic properties that demonstrate knowledge by reasoning. One of the important features of this chapter is the abstraction refinement method that reduces the verification time and memory usage for temporal-epistemic properties. Chapter 6 describes the opacity verification method on Datalog-based policies and experimental results, and Chapter 7 concludes the thesis.

1.7 Notations

As the convention and for the rest of this thesis, we write constants in `typewriter` font, variables in *italic* and key words in sans serif.

1.8 Publications

The thesis is partly based on the following publications:

- Moritz Y. Becker and Masoud Koleini. Opacity analysis in trust management systems. In *14th Information Security Conference (ISC 2011)*, 2011
- Masoud Koleini and Mark Ryan. A knowledge-based verification method for dynamic access control policies. In *ICFEM 2011: Proceedings of 13th International Conference on Formal Engineering Methods*, 2011
- Masoud Koleini and Mark Ryan. A knowledge-based verification method for dynamic access control policies. Technical report, University of Birmingham, School of Computer Science, Available at: `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/`, 2010
- Masoud Koleini, Hasan Qunoo, and Mark Ryan. Towards modelling and verifying dynamic access control policies for web-based collaborative systems. In *W3C Workshop on Access Control Application Scenarios*, 2009

A journal version of the paper “Opacity analysis in trust management systems” is also submitted to the *Journal of Information Security.*

CHAPTER 2

RELATED WORK

In this chapter, we briefly provide the definitions, features and categories of access control systems. We then introduce model-checking techniques as the formal verification method, which we will use in this thesis for modelling and verifying access control systems. Finally, we will explain abstraction and refinement in model-checking. Abstraction techniques enable us to reduce the size of state transition system and improve time and memory efficiency in model-checking. We also review several major works in the field of access control policy verification.

2.1 An overview of access control

Access control is the process of mediating the requests for accessing data in a system and determining whether the request should be granted or denied. Access control can be divided into three control categories [57]:

1. *security policy*: At the top of the access control is security policy, the high level description of the conditions and rules under which a user or process can access some resources in the system. Policies are in general dynamic and possible to change by the administrators when some requirements in the system are changed.

2. *security mechanism*: Access control mechanism enforces the policy through translating the requests into system acceptable structure.

3. *security model*: Access control model, which formally presents how the policy is enforced in the system, provides the link between the policy and the mechanism. In general, access control models are divided into two major categories of *discretionary* and *non-discretionary* access control which will be discussed later. Non-discretionary models also contain two major reference models of *mandatory* and *role-based* access control models.

Several important features that need to be included in access control systems are highlighted in [38, 57]. Some of the features are as follows:

- *Conditional authorizations*: Access permissions are granted or denied if some conditions in the system holds. The conditions can be in the form of system predicates.
- *Support for fine-grained and course-grained specifications*: Access control should support fine-grained authorization rules to be applied by the administrator in the system. However, administrating fine-grained access control is difficult and error prone. Therefore, access control should provide the support for administrating the authorization of groups of users and resources, which is the main motivation of designing role-based access control model.
- *Separation of duty*: This principle does not provide the sufficient authorization for the individuals in the system to perform fraudulent actions [84].
- *Delegation of authority*: Access control should provide the possibility of passing authorizations between agents. Delegation of authority enhances scalability and flexibility, but increases the complexity of access control.
- *Least privilege*: This principle states that each individual should have the minimum required permissions to perform his tasks. Least privilege in general is difficult or costly to achieve [57].

We now introduce major access control security models in the following sections.

2.1.1 Access control matrix

The concept of access control matrix was first introduced by Harrison et al. [52]. In their formal model of protection systems, a *configuration* is a triple (S, O, P) where S is the set of *subjects* (the entities that perform actions in the system), O is the set of *objects* (resources in the system) and P is the access control matrix that contains a row for each subject in S and a column for each object in O. The authors have considered $S \subseteq O$. Let R be the set of *generic rights* is the system, for instance, read, write and execute permissions in Unix-like file systems. If $s \in S$ and $o \in O$, then we have $P(s, o) \subseteq R$.

In [52], *commands* contain the operations that are able to modify the contents of access matrix. But in general, an access control matrix by itself does not provide a complete view of security policy. This is because the matrix does not model the operations that change the rights in the matrix.

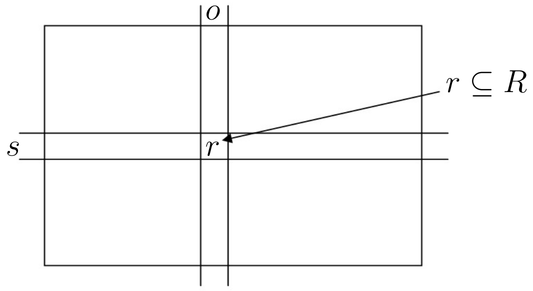

Figure 2.1: The general structure of an access matrix. r indicates the access rights that s has on o.

Access control lists (ACL) are the list of permissions attached to an object can be modelled by access matrices. Most of the operating systems have their own access control lists implemented in their file system, mainly known as *access control entries (ACE)* .

2.1.2 Discretionary access control

In discretionary access control (DAC), the *owner* of an object or the authorized entity decides about access permissions of the object. Therefore, access permissions are not regulated by the organization policy or rules. In DAC, owners can delegate the control or pass the permission of accessing the resources to other entities.

A security policy based on DAC can be presented by an access control matrix (defined in section 2.1.1). One of the problems of DAC policies is their large memory for storage and complicated administration. An example of DAC access control is file system permissions in Unix-like operating systems. In such systems, each file has an owner that determines the read/right/execute rights for the owner/group/other entities. Such access control is not fine-grained, but is simple to manage by individuals[1].

Discretionary access control suffers from several weaknesses like unauthorized information flow (Alice grants Bob read access to her file, Bob copies the content of the file into another file of his own and allows some unauthorized users to access the content of Alice's file), vulnerability to Trojan horses, unrestricted information usage and possibility of violating organization policy [57].

[1] Linux kernel version 2.6 allows users to apply more fine-grained access control lists using the command `setfacl`.

2.1.3 Mandatory access control

In contrast with DAC where the owners have the ability to override the permissions of their own objects, in mandatory access control (MAC) it is a central authority that enforces authorization rules to the subjects and object in the system. One of the most well-known examples of MAC is *multi-level security* (MLS) also known as BellLaPadula model [18] developed for military applications. MLS assigns security levels to the objects beginning from *top-secret* (most sensitive), *secret*, *confidential* and ends with *public* or *unclassified* (least sensitive). The subjects are also assigned with similar security levels. Bell-LaPadula model enforces two mandatory access rules:

- *No read-up*: A subject with a specified security level can not read the object with higher security levels. For instance, an entity with security level confidential can not read a document labelled with secret.

- *No write-down*: A subject with a specified security level can not write over the objects with lower security levels. This rule is also known as *-property.

The model also defines *strong *-property*, where a subject can write only over the objects of the same security level.

Security Enhanced Linux (SELinux) is the commercial implementation of mandatory access control in Linux distributions[1] supported by *National Security Agency*. SELinux applies least privilege principle to the system and server in such a way that the programs have the minimum required privileges to perform their task. This feature prevents the programs to harm the whole system if they get compromised.

2.1.4 Role-based access control

The main motivation for designing role-based access control model (RBAC) is to facilitate the administration in medium to large-scale multi-user systems [87]. RBAC as a form of non-discretionary access control first introduced by Ferraiolo and Kuhn [47] and officially maintained and developed by NIST (National Institute of Standards and Technology). Sandhu et al. [87] introduced four conceptual reference models. RBAC$_0$ is the basic model containing the minimum requirement for systems supporting RBAC. RBAC contains three core set of entities: *users*, *roles* and *permissions*. Users are in general human-beings, roles are job titles, responsibilities and ranks in the organization that are extractable from organization documents and charts, and permissions are the conditions under which the roles can access objects and resources. The permissions are application-specific, like

[1] SELinux is integrated in Linux kernel version 2.6.

read/write/execute in a file system or issue prescription, read prescription and read patient personal information in a healthcare system. The key feature of RBAC is the two relations of *user assignment* (UA) and *permission assignment* (PA). Users can dynamically be allocated to a set of roles, a role can be related to a group of users, a role can have several permissions and permission can be shared between a group of roles. The assignment of users to roles and roles to permissions can be changed dynamically by the administrator and changes in organization roles and policy.

Another component of RBAC is the *session*. Each session in the system is mapped to a user and to a group of roles that user has activated. In the case that more than one role are activated in a session, the permission is the union of the permissions of the activated roles. In RBAC$_0$, there is no restriction for the user to activate a subset of roles he belongs to simultaneously and still the principle of least privilege applies. That means the permissions of the user is the union of the permissions of the invoked roles, not all the roles that he is assigned.

RBAC$_1$ extends RBAC$_0$ by introducing role hierarchies. In RBAC$_1$, role hierarchy is a partial order over the set of roles with seniority relation. A *senior role* inherits the permissions of the related *junior roles*. The anti-symmetric property of the partial order prevents two roles to inherit from each other at the same time. Therefore, if a user is assigned to a role, he is implicitly assigned to all the corresponding junior roles.

The only difference between RBAC$_2$ and RBAC$_0$ is the application of constraints over the values. For instance, it should be impossible to allocate a user to mutually exclusive roles, as it may raise the risk of fraud in the system. Therefore, RBAC$_2$ supports separation of duties by applying the constraints to user assignments and permission assignments. *Cardinality constraints* like restricting the number of roles a user or a permission can be assigned is another way of maintaining organization's discipline, which is supported in RBAC$_2$ model. RBAC$_3$ combines the features of role hierarchy and constraints.

Sindhu et al. in [85, 86] proposed an RBAC-based model called ARBAC97 (Administrative RBAC 97) for administrating RBAC. Their model simplifies the administration of the systems with thousands of users and roles in a decentralized way. SARBAC [35, 36] modifies ARBAC by defining administrative functions in terms of administrative scope, which is used to control the user to role and role to permission assignments.

Role-based access control is actively implemented and used in various products like IBM WebSphere InterChange Server, IBM AIX operating system, Sun Solaris and Microsoft Exchange Server.

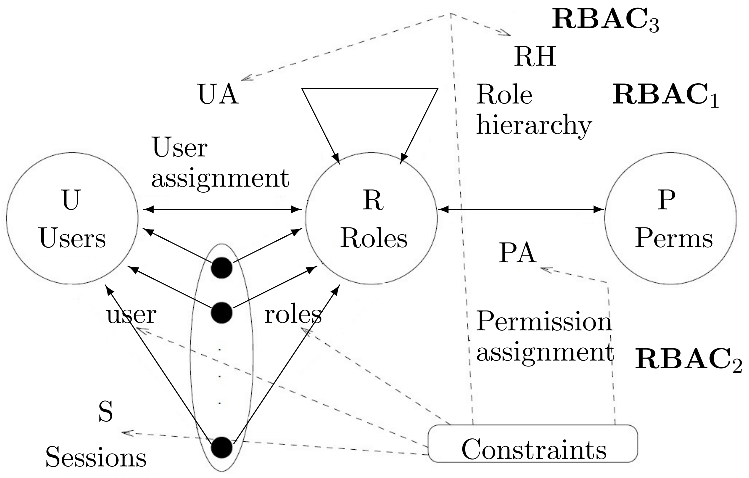

Figure 2.2: RBAC reference models [87].

2.2 Access control policy

An *access control policy* is a set of *rules* that is written in a formal *policy language* [10]. The rules express the regulations that should be enforced and the policy language needs to be flexible and expressive enough to accommodate the common requirements (also known as *policy idioms* [10]) like constraints, delegation of authority, separation of duty and role hierarchy.

Policy languages can provide *language constructs* to facilitate the definition of requirements and constraints. For instance, some languages use the constructs that facilitate encoding the policies of role-based systems [63, 79]. In addition, [79] supports the definition of static and dynamic separation of duty and cardinality constraints in the policy. SecPAL [13] uses the constructs *says* and *can say* to support digital signature in a decentralized networks and delegation of authority. SPL [83] is a policy language that is specifically designed for expressing various constraints of type history constraints, enforcing and expressing obligations, and invariant constraints.

One of the important classification of access control policies is *dynamic* (or *state-based*) and *static*. In dynamic policies [11, 94, 82, 42], performing actions depend on the authorization states and action performance results in changing the authorizations. Static policies [77, 13, 70, 48] express the conditions where access request, which can be complicated and contain lots of dependencies, are granted or rejected in a specific authorization state. Static policies (simply consider an access matrix) does not provide the details of the actions, and access requests does not change the authorization states.

Now, we will discuss different languages for access control policies and explains their specifications, advantages and disadvantages.

2.2.1 eXtensible Access Control Markup Language (XACML)

The *eXtensible Access Control Markup Language* (XACML) [77] is an XML-based policy language approved by the OASIS committee. The main motivation for defining a standard for authorization languages is that lots of application -specific policy languages are designed, but the authorization rules can not be shared between different applications. XACML acts as a common language for the applications to interact and share their authorization rules[1]. The policy language is flexible enough in the context of extensibility and can be *extended* to accommodate application-specific requirements.

Security policy in general is separated from enforcing the decisions. In XACML data-flow model, policy is created and stored in *policy administration point* (PAP). When an authorization request is submitted, *policy decision point* (PDP) renders access decisions. Access control decisions are enforced by *policy enforcement point* (PEP), and *policy information point* (PIP) acts as a store of resource attributes and returns the required information to PEP. The data-flow model contains the entity *context handler* with the duty of translating decision requests in native format to XACML and translating authorization decisions in XACML to the native form.

The data-flow in XACML is as follows: PAP makes the policy accessible to PDP. Access requests are submitted to PEP. PEP sends the access requests to context handler, including the attributes (characteristics) which is passed to PDP after translating to the XACML request. If PDP requires additional attributes, it sends the request to the context handler. Context handler collects the required attributes (optionally including the resources) from PIP and sends them to PDP. PDP evaluates the policy, prepares the response including the authorization decision and send them to context handler. Context handler translates the decision back to the native form and sends it to PEP, which enforces the decision. In the case that access is granted, PEP permits access to the requested resources.

XACML language model v3.0 is composed of three top-level components *rule*, *policy-set* and *policy*. Rule is the basic unit of the XACML language with the main components: a *target*, an *effect*, a *condition*, *obligations* and *advice*. target defines the decision requests where the rule applies. The effect of a rule is the intended decision that should be enforced if the rule is evaluated to true, which is always *permit* or *deny*. Condition is a Boolean expression over the predicates implied by the target, which refines the applicability of the rule. The rule is *applicable* if both the target and the condition evaluate to true, and then the effect will be returned.

The policy in XACML is composed of the main components: a target, a *rule-combining algorithm*, a set of rules, obligations and advice. Obligations are included in the context

[1] XACML v3.0 was approved by OASIS in 2009.

which is returned by the PDP to PEP (through context handler) after evaluating the rules. Similar to obligations, advice is also included in the context returned by PDP, but in spite of obligations, it can be ignored by PEP. Rule-combining algorithm specifies the procedure of combining the result of evaluating the rules in the policy. In general, it handles conflict resolution in the cases that several rules in the policy are applicable.

The policy-set contains a target, a *policy-combining algorithm*, a set of policies, obligations and advice. Similar to rule-combining algorithm, the policy-combining algorithm define the procedure of combining the results of verifying the included policies. Obligations and advice is defined similar to the policy.

Combining algorithms (rules and policies) divide into four categories. In *divide overrides*, if at least one of the rules or policies evaluates to deny, then the result is deny. In *permit overrides*, the result is permit if some rules or policies return permit. The algorithm *first-applicable* evaluates to the result of the first rule or policy, which is applicable to the decision request. *Only-one-applicable* applies only to the policies, and if only one policy or policy-set is applicable in the context of a target, it evaluates to the result of that policy. Otherwise, it evaluates to *not applicable* if no policy or policy-set is applicable, and evaluates to *indeterminate*, if more than one is applicable.

XACML policy language suffers from several weaknesses. The policies written in XACML are verbose and complex. They are hard to read by someone who is not familiar with the rules and difficult to analyse. Interactions between the main components like PEP and PDP are not standardized and policy administration in XACML is not modelled or discussed.

2.2.2 Role-based trust management framework (RT)

The term *trust management* was first introduced by Blaze et al. [21]. The trust management problem deals with the following question which is also known as *proof-of-compliance* problem:

Given a request to perform a specific action and a set of *credentials* (signed by different authorities), does the request comply with the local policy?

In [21], the authors argue that the simple name to certificate binding does not provide enough security in terms of legal actions (names to actions mapping problem). PolicyMaker [22] uses a trust management engine over the submitted *credentials* in order to process the authorization *queries*, which are the requests by one or a sequence of entities (public keys in the context of decentralized trust management system) to perform an action regarding to the local policy. The queries in PolicyMaker are of the form:

$$key_1, \ldots, key_n \textsf{ Requests } Action$$

The queries are processes based on the *assertions* which contain the trust information. Each assertion contains a *source* which is the local policy or in the case of signed assertions, public key of a third authority, an *authority structure* which is a sequence of public keys representing identities whom the assertion applies, and a predicate called *filter*:

$$source \textsf{ Asserts } authority\ structure \textsf{ Where } filter$$

KeyNote [20, 19] is the successor of PolicyMaker. The advantages of KeyNote over its ancestor is simpler C-like syntactic notations for the predicates and assertions, expressiveness in terms of delegation of trust, and extensibility while preserving the compatibility with PolicyMaker.

Role-based Trust-management framework (RT) [70] is logic-programming-based policy languages which combines trust management with role based access control (RBAC). The delegation of the authority in PolicyMaker and KeyNote is restricted. For instance, a book store can not simply specify the policy statement "anyone who is a student is entitled to discount" [68]. The solution is the delegation of the discount permission by the book store to the university, and then explicitly delegation of the permission by the university to each student's key. The above approach makes the access control system inefficient and difficult to manage. To overcome such limitations, RT uses Delegation Logic [68] which is specifically designed to facilitate expressing delegation of authorities.

RT has a family of languages: RT_0 is the basic language where the roles are simply the names without any arguments, RT_1 expands RT_0 by adding parameterized roles, RT^T adds the construct *manifold roles* and role-product operator for expressing threshold and separation of duty policies. Delegation of role activations is supported in RT^D.

In RT, a role is named by a principal (identity which is identified by its public key) and a role term. For A as a principal and R as a role term, $A.R$ denotes the role R defined by A. The basic language RT_0 contains the following rules:

- **Simple member:** $A.R \longleftarrow B$ means B is a member of the role $A.R$.

- **Simple containment:** $A.R \longleftarrow B.R_1$ means the role $A.R$ contains all the principals of the role $B.R_1$.

- **Linking containment:** $A.R \longleftarrow A.R_1.R_2$ means the role $A.R$ contains the principals of $B.R_2$ (R_2 is the role defined by B) for every B that is a member of $A.R_1$.

- **Intersection containment:** $A.R \longleftarrow B_1.R_1 \cap \cdots \cap B_n.R_n$ means the role $A.R$ contains the principals that are the members of all the roles $B_1.R_1$,...,$B_n.R_n$.

The following extra rules define the simple and linking delegation and are definable by the above basic rules:

- **Simple delegations:** $A.R \Longleftarrow B : C.R_1$ where the part $C.R_1$ is optional. This rule means A delegates its authority over the role R to B. The optional part restricts B to the members of the role $C.R_1$. This rule is equivalent to $A.R \longleftarrow B.R \cap C.R_1$.

- **Linking delegation:** $A.R \Longleftarrow A.R_1 : C.R_2$ where the part $C.R_2$ is optional. This rule means that A delegates its authority over the role $A.R$ to the members of $A.R_1$. The whole rule can be written as $A.R \longleftarrow A.R_1.R \cap C.R_2$.

RT framework uses DATALOG$^{\mathcal{C}}$ (DATALOG with constraints) [69] for the deduction engine. DATALOG as a subset of PROLOG is a logic programming language without function symbols, with restricted use of negation and recursion (stratification restriction) and range-restricted variables. The query evaluation in DATALOG is sound and complete. The lack of function symbols disables DATALOG-based languages to express *structured resources*, but makes the language *tractable.* The constraints in DATALOG$^{\mathcal{C}}$ enable the trust management language designers to define access permissions over structured resources.

2.2.3 SecPAL

Similar to RT, SecPAL is a declarative authorization language based on DATALOG$^{\mathcal{C}}$. The major success of SecPAL is in its flexible delegation of authority, which allows defining unlimited delegation path in one *policy assertion.* The definition of constraints in SecPAL is unrestricted and constraints does not make the language intractable (In RT, constraint domains are not guaranteed to be tractable [70]). Although SecPAL does not allow negation in the assertions in order to prevent intractability and ambiguity, it permits negation inside the queries.

Abadi et al. [1] first used the term `says` in their access control calculus to denote the intention or signature of an agent over an assertion. The type of the assertion can be imperative or factual. For instance, `Alice says` "`Delete AliceSecret.wmv`" is an imperative assertion, while `TrustedParty says` "`Alice is the owner of AliceSecret.wmv`" is factual.

SecPAL uses the same notation "says" for issuing an assertion as in [1], and uses the term "can say" for delegation of authority, which is similar to *controls* in [1]. The following is a part of a local policy for a conference paper review system, where `TTP` is a trusted third party and CPRS is a particular conference paper review system (like easyChair or HotCRP):

TTP says Alice is a chair (1)
CPRS says TTP can say x is a chair (2)
CPRS says x can allocate y as the reviewer of p if (3)
  x is a chair,
  y is a PC member,
  p is a paper
CPRS says x can allocate y as the sub-reviewer of p if (4)
  x is the reviewer of p

The above is an example of a decentralized conference paper review system, where the chair submits assertion (1) to the conference system in order to prove his identity. SecPAL uses *can act as*, for principal aliasing. Consider the following assertion:

CPRS says x can act as proceeding author if (5)
  x is an author,
  currentTime() $\leq$ 12/3/2011

The assertion states that all the facts that apply to a proceeding author also applies to a principal which is an author and before the specified deadline. The general form of an assertion is of the form A says $fact$ if $fact_1, \ldots, fact_n, c$ where A is the principal who issues (or digitally signs, in the context of distributed systems) the assertion, facts specify properties over the principals and c is constraint. Constraints (as currentTime() $\leq$ 12/3/2011 in the above assertion) contain equality (=), numerical inequality ($\leq$), regular expressions (r matches *pattern*), negation and conjunction of constraints. Inequality and disjunction can be expressed combining basic constraints. In general, the expressiveness of SecPAL policy language is the result of its flexible and supporting class of constraints.

SecPAL can act as the engine of a *reference monitor*, which validates access requests to the resources and enforces the policy. A principal sends a set of assertions (also known as *credentials*) and then requests access. SecPAL evaluates the query against the union of local policy and submitted assertions and verifies if the access is granted or rejected. While negation in recursive policies may result in complexity and undecidability[1], SecPAL allows negation in the authorization query. Queries can contain constraints in the form of the ones used in assertions, conjunction, disjunction, negation and existential quantification.

Authorization query table introduced by SecPAL which provides a mapping between

[1] Some declarative languages like PROLOG define negation in the form of *negation as failure*. PROLOG derives *not* p if it fails to derive p.

parameterised access requests and authorization queries in SecPAL query language. For instance and in a file sharing server, $\texttt{read}(x, f) \mapsto$ FileServer says x can read f is the mapping from the request for reading file f by the principal x to the query language.

Using authorization query table preserves separation of duty in declarative languages like SecPAL which does not let negation in the assertions. Negation plays an important role in preserving separation of duties. Let's consider assertion (3). Integrity constraints in the system do not let the author to be assigned as the reviewer of his paper. Therefore, we would like to have $\neg(x$ is the author of $p)$ as a fact, while negation is not allowed in facts. Authorization query table provides the support for such cases. So, the request for adding a principal as the reviewer will be mapped to the query in the following way:

$\texttt{addReviewer}(x, y, p) \mapsto$ (6)

CPRS says x can allocate y as the reviewer of p,

not (y is the author of p)

SecPAL can easily express different access control models like discretionary access control (DAC):

Bob says Alice can read f if (7)

f is a file, Bob is the owner of f

and mandatory access control (MAC):

FileServer says *user* can access *file* if (8)

user is a manager, *file* $\preceq$ ManagementDir, isConfidential($file$)=TRUE

where $\preceq$ is the path constraint and shows that the directory ManagementDir includes the *file*.

Role hierarchies can be easily explained by using the keyword *can act as*. The assertions and queries will be translated into $\text{Datalog}^{\mathcal{C}}$ and then evaluated.

SecPAL and RT are the two well-known authorization languages based on $\text{Datalog}^{\mathcal{C}}$ and are categorized as stateless policies. Given a policy and a set of submitted credentials, their trust management engine decides whether a request should be granted or denied. This thesis focuses on finding vulnerabilities over access control systems. As a part of our research, we will try to find if a set of available credentials and queries for a principal enables him to infer some confidential information contained in the policy. We will formalize such attacks (called *probing attacks*) and their detection algorithm in chapter 6.

2.2.4 RW

As reviewed in section 2.2.1, XACML is a standard language designed for access control policy definition. Zhang et al. [94] introduced a framework that has its own policy language and synthesising mechanism, but is able to translate the policy written in its formal language into XACML. The modelling formalism is called RW and is supported by a model-checking tool called AcPeg (access control policy evaluator and generator). The formal verification support provides the opportunity to ensure that first, legitimate properties hold in the policy which means the users have enough permission to carry out the required actions, and second, malicious behaviour is prohibited. In spite of the existence of various model-checking tools like NuSMV [27, 26], Alloy [59, 60] and SPIN [56], RW has its own model-checking mechanism.

RW formalism uses propositional variables which are Boolean variables. the policy language allows defining two classes of *parameterized rule-definitions* over predicates: *read* and *write*. *Read* rules define the permissions for reading the truth values of instanced predicates, and write rules define the permissions for overwriting their values. For example in a conference paper review system, the following fragment of the policy specifies the condition in which *user* (the agent that performs reading or overwriting) can read the value of $\texttt{reviewer}(p, a)$ and the condition he can overwrite the value (assigning a principal as the reviewer of a paper) [94]:

$$
\begin{array}{l}
\texttt{reviewer}(p, a)\{ \\
\quad \texttt{read} : \texttt{pcmember}(\mathit{user}) \wedge \neg\texttt{author}(p, \mathit{user}) \\
\quad \texttt{write} : (\texttt{chair}(\mathit{user}) \wedge \texttt{pcmember}(a) \wedge \neg\texttt{author}(p, \mathit{user})) \vee \\
\qquad (\texttt{pcmember}(\mathit{user}) \wedge \mathit{user} = a \wedge \texttt{reviewer}(p, \mathit{user})) \wedge \neg\, (\exists b\, \texttt{subreviewer}(p, \mathit{user}, b))
\end{array}
$$

The rules in the policy specify dynamic state changes when a read or write is performed. Overwriting action changes the state of the system and therefore, changes access permissions for other agents.

The model-checking algorithm in RW checks if a property (or *goal*) is achievable by a *coalition of agents* and through a sequence of reading/overwriting actions in the model build based on the policy. A coalition is a set of agents which co-operate together in order to achieve the goal. In the case that the goal is achievable, the model checker produces a sequence of actions that leads the coalition to the goal and is called *strategy*.

Transition system: The states in RW are called *knowledge states* where each state accumulates the initial knowledge of the coalition and the knowledge gained by sampling or overwriting the propositions when executing a strategy. If the value of a proposition is overwritten by an agent, the agent learns the current value of that proposition and he does not need to sample its value in future steps. Reading the truth value of a proposition also adds the knowledge about the current and initial value of that proposition. For each proposition p, four Boolean knowledge variables v_{0p}, t_{0p}, v_p and t_p are used: v_{0p} is true if the initial value of p is known by the coalition, t_{0p} stores the initial value of p when v_{0p} is true, v_p is true if the current value of p is known by the coalition and t_p stores the current value when v_p is true. If P is the set of propositions, the knowledge state is defined by (V_0, T_0, V, T) where $V_0 = \{p \in P \mid v_{0p} = \top\}$, $T_0 = \{p \in P \mid t_{0p} = \top\}$, $V = \{p \in P \mid v_p = \top\}$, $T = \{p \in P \mid t_p = \top\}$ and the transitions are as follows:

$$
\begin{aligned}
&(V_0, T_0, V, T) \xrightarrow{\text{sampling } p \text{ returns } \top} (V_0 \cup \{p\}, T_0 \cup \{p\}, V \cup \{p\}, T \cup \{p\}) \\
&(V_0, T_0, V, T) \xrightarrow{\text{sampling } p \text{ returns } \bot} (V_0 \cup \{p\}, T_0 \cup \{p\}, V \backslash \{p\}, T \backslash \{p\}) \\
&(V_0, T_0, V, T) \xrightarrow{p:=\top} (V_0, T_0, V \cup \{p\}, T \cup \{p\}) \\
&(V_0, T_0, V, T) \xrightarrow{p:=\bot} (V_0, T_0, V \backslash \{p\}, T \backslash \{p\})
\end{aligned}
$$

The first two transitions are the result of sampling proposition p and the last two are the transitions made by overwriting p. Sampling p is only permitted when the value of p is not known ($p \notin V_0$). If K_G represents the knowledge states in which the coalition "knows" that the goal is achieved, then a strategy is a sequence of sampling/overwriting steps that lead the coalition from initial knowledge states to K_G. In strategy finding algorithm, it is assumed that an agent performs an action if he knows that he has the right permission.

Constraint definition in RW is flexible by allowing negation and universal and existential quantifiers in permissions. But the framework suffers from several major weaknesses:

- Overwriting steps are able to update only one proposition. This weakness reduces the expressiveness of the language as some scenarios require updating several propositions in one step. For instance, if Alice is a reviewer in a conference paper review system, when she resigns as the reviewer of a paper, all the sub-reviewers she allocated to the paper should get deleted at the same time. Such bulk updating rules can not be specified in RW policy language.

- For each proposition p, there are 7 relevant valuation of knowledge variables. Therefore, the total number of knowledge states increases by the factor of 7 when the number of propositions increases which causes the state explosion even in small and medium size models.

- The knowledge state in RW stores the history of reading or altering propositions. The transition system shows that the knowledge is incremental during state transitions. Knowledge variables for different propositions are also independent. A side effect of such an approach is that reasoning about knowledge is not possible. For example, consider the case that whenever the variables p is true, variable q turns to true. Then knowing that p evaluates to true should reveal the value of q, while RW is unable to handle such reasoning.

This thesis addresses the weakness of RW by first introducing a policy language that supports variable bulk update and corresponding verification framework. The framework performs verification over system states and is able to verify knowledge gained by reading system variables. Our method does not handle memoryful knowledge as in RW, but the ability of the language to handle variable bulk update provides the potential of incorporating memory into the system states. Chapter 4 explains the framework and implemented tool. To verify knowledge by reasoning, we have proposed another verification method, which is described in chapter 5.

2.2.5 DynPAL

DynPAL is a dynamic authorization policy language designed by Becker [11]. Comparing to RW, DynPAL provides the additional features of bulk updates, nested actions, intermediate conditions and postconditions. One of the major features in DynPAL is that variables may range over infinite domains as in decentralized systems, the number of principles may be unbounded. Two analysing methods are proposed to verify *reachability* and *safety* properties over the policy. Reachability deals with the problem of finding a sequence of actions that lead to a state that satisfies the required property, beginning from initial states. Safety is the complimentary problem: the states that satisfy an unwanted property are not reachable from some initial states. To verify the reachability of a property over a policy in DynPAL authorization language, the policy and the query are translated into the PDDL (Planning Domain Definition Language) [49, 92] which is the standard artificial intelligence planning language, and verified by an AI planner [55]. In the case of reachability analysis, the variables as the arguments of predicates range over finite domains, otherwise the problem is undecidable. When analysing safety properties (*policy invariants*), the policy and invariance hypothesis can be transformed into a problem of first order logic (FOL) and solved using a first order logic theorem prover [81].

In DynPAL, a state is a set of *extensional ground atoms* known as *extensional database* in Datalog. Actions in DynPAL change the state by adding some ground atoms into the state or retracting some of them. For instance and in our example of a conference paper review system, the following rule:

$\texttt{delReviewer}(p, a) \leftarrow \texttt{reviewer}(a), \neg\texttt{submittedReview}(p, a),$

$-\texttt{reviewer}(p, a), -\{\texttt{subReviewer}(p', a', b) : p' = p, a' = a\}$

demonstrates the situation where a principal resigns as the reviewer if he has not submitted his review yet. The effect of such resignation is the retraction of the reviewer and his allocated sub-reviewers form the extensional database and therefore it changes the state. In DynPAL, insertion or retraction of atoms into the state are executed from left to write. In the above rule, sub-reviewers are retracted after the reviewer. In some cases, this sequential execution may result in different states when the order of updates is changed. For example after executing $-p(0), +p(0)$, the atom $p(0)$ will stay in the state, while it will not appear in the state when $+p(0), -p(0)$ is executed.

Becker in [11] argues that specifying an initial state for the verification of safety properties is a limitation. While Becker's approach for evaluating the reachability of a property requires a single initial state to be specified, his approach for evaluating a safety property does not have such a requirement. Experimental results show that in the case of verifying safety properties, theorem proving and model-checking may have better performance than a planner. Planner performs considerably faster when a plan to the states that satisfy the property exists.

Introducing only one single initial state is rather restrictive but eliminating the requirement to specify the initial condition seems to be too liberal. In practice, we would like to verify if all the states that are *reachable from the states that satisfy the initial condition* also satisfy the safety property. Therefore, it is more desirable to consider a set of initial states instead of a single state. Moreover for such definition of safety, the theorem proving approach may produce false-negative results in some scenarios as the states satisfying safety property may be legitimately reachable from some system states other than the initial states. Comparing to our work in this thesis, DynPAL is unable to evaluate dynamic policies against information leakage vulnerabilities as a result of reasoning.

2.2.6 Deontic logic for privacy policy

Aucher et al. [7] developed a framework for security policies to specify and reason about epistemic properties and check if they comply with the policy. The *privacy policies* are defined in terms of permitted or forbidden knowledge. A method for reasoning about privacy policies using an extension of a modal logic framework for security policies is provided, which also enables reasoning about confidentiality by expressing epistemic modalities [37]. Their approach uses deontic logic [90] with obligatory and permission modalities. In deontic logic, the notation $O\alpha$ means "it is obligatory (or it ought to be) that α", and $P\alpha$ means "it is permitted that α". Aucher et al. introduced dynamic and epistemic features to the previously developed modal logic and proved the soundness and completeness of the new logic, named DEDL (Dynamic Epistemic Deontic Logic). The paper deals with two agents as *sender* and *receiver* which communicate together. If sender transfers some information to the receiver, then receiver *knows* the information. The paper deals with verifying what information can be send from sender to the receiver so that the receiver will not be able to use them to reason about confidential information in sender's side, or what information is required for the receiver in order to know the obligatory information. The language uses modality operator $O_s\alpha$ to say "it is obligatory for the sender that α", $K_r\alpha$ to say "the receiver knows that α" and $K'_r\alpha$ similar as before for "the receiver knows that α" while the second one always places in the scope of obligation modality and is known as *ideal knowledge* of the recipient.

The dynamic part of the logic deals with sending or promulgating data. The language adds the properties that describe what happens after recipient learns a fact or sender promulgates some information. $[\mathit{send}\ \psi]\ \phi$ stands for "after recipient learns ψ, ϕ holds" and $[\mathit{prom}\ \alpha]\ \phi$ says "after sender promulgates α, ϕ holds.

The model in [7] only contains one sender and one receiver. Therefore, a multi-agent system and the knowledge gained by interaction of the agents cannot be modelled in their framework.

2.3 Model-checking for policy verification

Researchers may use different policy verification methods depending on their requirements, system properties and experience. Formal verification techniques contain the following main parts: (1) a formal *specification language* to describe the properties (2) a *model* that is presented by a *description language*, and (3) a *verification method* to determine if the model satisfies the property. The verification method can be divided into the two categories of *proof-based* and *model-based* techniques:

- The proof-based technique tries to find if the specification formula ϕ is *derivable* in a logical system specified by a set of rules and axioms.
- In a model-based technique, the system is presented as a model M, and verification computes whether the model satisfies the specification formula ϕ (denoted by $M \models \phi$).

Model-checking approach is simpler, easier to describe and automated for finite-state models. Other interesting features of model-checking are fast verification, independence of proof theory, producing counterexamples consisting of execution traces, and expressive logic for *concurrency* properties. The main disadvantage of model-checking that motivated us to propose a method for abstraction refinement is the *state explosion problem* which happens when the number of variables in the model increases.

The model-checking is based on verifying a *temporal property* over a model M which is a finite-state transition system. Temporal logic describes the rules and symbolisms for the presentation and reasoning about the properties in terms of time, i.e. the properties that can be true in some states and false in some other states.

2.3.1 Linear-time and branching-time temporal logic

The specification languages in general fall into two categories of logics: *linear-time* logics and *branching-time* logics. In linear-time logics, time is thought to be a set of paths of time instances. Branching-time logic considers the time to be as a tree that branches from current time to future.

Linear-time temporal logic (LTL) contains the *modalities* (also known as *connectives*) that refer to the time in future. The syntax of LTL is built over the propositional atoms, logical operators and modality operators X (neXt state), F (some Future state), G (Globally or all future states), U (Until), R (Release) and W (Weak-until). It is easy to show that the {X, U} is an *adequate set* of modalities, meaning that the other modalities can be expressed in terms of X and U. As an example for LTL logic, in a microwave oven it is impossible to get a state where the microwave is working and the door is open. This property is specified in LTL as $G\neg(\texttt{started} \wedge \texttt{doorIsOpen})$. A state s in a model M satisfies an LTL property if the property holds on all the paths that begin from s.

Computation Tree Logic (CTL) is a branching-time logic that allows existential and universal quantifiers over paths. From the point of quantification over paths, CTL provides more flexibility for defining specifications. On the other hand, LTL allows selecting a range of paths by describing those paths with a formula, which is not possible in CTL. CTL keeps the modalities U, F, G and X of LTL and adds universal path quantifier that expresses "for all paths" and existential quantifiers that means "a path exists". For example $AG(p \rightarrow EFq)$ expresses the property that for all paths and for all the states along the paths (denoted by G), if p holds in a state, then there exists a path that a state along it satisfies q. CTL* is an expressive logic that combines the expressive powers of CTL and LTL.

NuSMV (New Symbolic Model Verifier) [27, 26] is a well-known model-checking tools which verifies the properties of type LTL and CTL. NuSMV accepts a model written in its description or modelling language together with some specifications and checks if the model satisfies the specifications. Another well-known model-checker for LTL is SPIN [56]. The name SPIN stands for "Simple Promela (Process Meta Language) Interpreter".

SPIN uses *Büchi automata* as a part of model-checking algorithm.

A part of this research (chapter 4) deals with the properties that can not be expressed in CTL or LTL. Therefore, we need to implement our own model-checking tool to verify such properties. In chapter 5, we express our properties in CTLK, which is CTL logic integrated with knowledge modality (refer to section 2.3.3 for more information).

2.3.2 Alternating-time temporal logic

Alur et al. [5] introduced *alternating-time temporal logic* (ATL) which generalises branching-time temporal logic. ATL allows selective path quantifiers and defines a natural language for *open systems* while LTL and CTL are specification languages for *closed systems*. In closed systems, the system behaviour is determined by its own internal state, but in an open system, the interaction between the external environment and the system affects the behaviour of the system. Besides existential and universal quantifications over computation paths, ATL deals with the question "can the system resolve the internal state in such a way that the satisfaction of the specification in guaranteed, no matter how the environment reacts?" [5].

ATL is suited for multi-agent or multi-process distributed systems as a *concurrent game structure*. Each state transition is the result of the combination of movements of the agents in each (time) step. To compare the properties in CTL and LTL, let's assume a property for the cache in a multi-processor system which states that deadlock for the processor a should never happen (a cache-coherence property). The property in CTL can be specified as (in modal logic notation): $\forall\Box(\exists\Diamond\texttt{read} \wedge \exists\Diamond\texttt{write})$. The property says "is it possible for all the processors to *collaborate* so that the processor a can eventually read and write" which is called *collaborative possibility*. The ATL formula is of the form $\forall\Box(\langle\langle a\rangle\rangle\Diamond\texttt{read} \wedge \langle\langle a\rangle\rangle\Diamond\texttt{write})$. The property specifies "always the processor a can eventually access the memory, no matter what other processors do" which is known as *adversarial possibility*.

Mocha [3, 2, 4] is the model-checker for ATL which supports *modular* specifications, reasoning about synchronous and asynchronous heterogeneous systems, system execution simulation using randomization and manual techniques and requirement verification. Some early experiences show that the capability of Mocha in verifying large systems is limited [93].

2.3.3 Model-checking epistemic properties

One of the most commonly used approaches in the concept of logic of knowledge is $KT45^n$ or in some resources, $S5_n$. Modal logic of knowledge $KT45^n$ is generally used in *multi-agent systems*, where each agent has its own knowledge about the world. A multi-agent system contains a fixed set of agents. The modality K_i where i is an agent denotes the knowledge of the agent i. For example, $K_1 p \wedge K_1 \neg K_2 p$ means that agent 1 knows p, and also knows that agent 2 does not know p. $KT45^n$ also introduces modalities E_G that means everyone in group G knows and C_G as the *common knowledge* that means everyone knows, and everyone knows that everyone knows, and everyone knows that everyone knows that everyone knows, and so on. *Distributed knowledge* D_G means the knowledge is distributed among the members of the group and they can work the value out together if they do not have the knowledge individually.

The model that linear-time and branching-time temporal logic will be evaluated on is a Kripke model [58], which defines temporal transitions between the states. *Interpreted systems* [45, 46] are the state transition models with one local state assigned for each agent. Interpreted systems are specifically designed to reason about distributed systems in terms of knowledge. Interpreted systems are multi-agent frameworks where the global states are the Cartesian product of the local states, and the local states represent the accessible information for the agents. The system is synchronised with an external clock. In each clock cycle, each agent submits an action that is permitted according to the local state he is in (determined by the concept of *protocols*) and the *joint action* is the Cartesian product of the actions each agent submit. We will discuss the framework in detail in future sections and when we model reasoning about knowledge in access control systems.

One of the well developed model-checkers that evaluates knowledge-based properties over interpreted systems is MCMAS (Model-Checker for Multi-Agent Systems) [75, 72]. MCMAS accepts an input script file containing the model in a description language called ISPL, together with the specification. The specification formula is in ATLK (Alternating-time temporal logic with knowledge). As an example and in the bit transmission problem [45], a sender sends the value of a bit over a noisy channel that may drop the message, but does not tamper it. In the case that the message receives the other end of the channel, receiver replies by sending back an acknowledgement. Therefore if the sender receives the acknowledgement, it knows that the receiver knows the value of the transmitted bit. This property is expressed in MCMAS by the following syntax:

```
AG(recack -> K(Sender,(K(Receiver,bit0) or K(Receiver,bit1))))
```

where `recack` is the proposition that shows the acknowledgement has received to the sender and `K(x,y)` means `x` knows that `y`.

The main trade-off for the expressiveness of the properties that can be verified in interpreted systems is the large number of state space and high verification time specially when verifying epistemic properties. Therefore to verify medium to large systems, we need to adopt abstraction techniques to overcome *state explosion problem*. One of the main contributions of the thesis is adopting abstraction techniques to verify temporal and epistemic properties over interpreted systems. We will review the common abstraction techniques in the next section.

2.4 Abstraction techniques

Clarke et al. introduced existential abstraction technique for model-checking large state-space systems [31]. The concept of abstraction for temporal-logic model-checking is build over the theory of abstract interpretation [34]. Introducing binary decision diagrams (BDD) [23] in 1986 improved the capability of verifying specifications over medium-scale finite-state models, but it was still unable to handle complex properties over large-industrial designs. Although the model-checking algorithm for verifying branching time temporal logic CTL [29] is linear in the size of transition system and the length of specification, the size of transition system increases exponentially when the number of variables increases. This problem is known as *state explosion problem* in model-checking.

While applying BDD techniques in 1990 significantly increased the size of models from 10^{20} to 10^{100} states [24], Clarke et al. in 1994 claimed that using abstraction techniques enabled them to verify large systems with 10^{1300} reachable states [31]. The first abstraction technique called *existential abstraction* [31] which overestimates the concrete model with the abstract one. CTL* is an expressive logic that combines the expressive power of branching-time and linear time logic (LTL) [80, 89]. In existential abstraction, if a property in ACTL* holds in the abstract model, then it holds in the concrete (original) one. ACTL* is the fragment of CTL* where only universal path quantifier and negation over atomic formulas is used.

In [30], Clarke et al. propose a complimentary approach for the existential abstraction. They used the feature of model-checking ACTL* that returns a *counterexample* in the case that the property does not hold in order to refine the abstract model. In existential abstraction, the states are partitioned into clusters that construct the states in the abstract model. For the refinement, the clusters split up into different sets to make the model more precise. The power of counterexample-guided refinement is the intelligent splitting of the clusters, in a way that the previous counterexample does not occur in the refined model.

2.4.1 Counterexample-guided abstract refinement (CEGAR)

Clarke et al. [30] proposed a method to refine an abstract *Kripke model* build by existentially abstracting the concrete model. They combined two techniques of *symbolic model-checking* [43, 28] and abstraction to achieve the best results for overcoming state explosion problem. Symbolic model-checking prevents explicit construction of the Kripke structure by encoding the set of states and transition relation into Boolean formulas. Set operations like union and intersection can be transformed into disjunction and conjunction of Boolean formulas. BDD techniques traditionally play an important role in presenting Boolean formulas.

Simulation relation relates the states in the concrete model with the abstract one. Let the relation $H \subseteq S \times S'$ be a simulation relation between the two Kripke models M and M' where S and S' are the set of states in M and M'. If $(s, s') \in H$ and there is a transition from s to $s_1 \in S$ in the model M, then *there exists* $s'_1 \in S'$ where s' has a transition to s'_1 and $(s_1, s'_1) \in H$. There are some other constraints that make a relation to become a simulation relation between Kripke models, but it will be discussed later in technical sections. If such a relation exists, we say that model M' simulates M denoted by $M \preceq M'$. The important property that will hold between the two models is that if φ is an ACTL* property over the atomic propositions of M', $M \preceq M'$ and $M' \models \varphi$, then $M \models \varphi$ (the notation $\models$ stands for the *satisfaction relation*). In practice, we use an abstraction *function* that maps the states in the concrete model to the corresponding states in the abstract model [30].

The process of counterexample-guided abstraction refinement consists of three steps: (1) generating the initial abstraction, (2) model-checking abstract model, (3) refining the abstraction. In a model described by a program like a hardware description language, the states are the different valuations of the variables. It is assumed that the number of variables and the domains in which the variables are associated are finite. Therefore, we will have a finite state model. To have a symbolic approach for clustering the states, the variable domains are split into the *variable clusters*. The initial abstracted model is built in such a way that it simulates the original one.

The result of model-checking the abstract model falls in one of the following: (1) the property holds in the abstract model, (2) the property does not hold in the abstract model and therefore a counterexample is generated. In case (1) and by the above discussions, if the property holds in the abstract model, it also holds in the concrete model. In the case that the model-checking results is an counterexample, it should be checked in order to find out if it corresponds to a counterexample in the concrete model or it is *spurious*.

Spurious counterexample identification: The counterexample generated by the verification of an ACTL* specification is either a *finite path* when a safety property fails

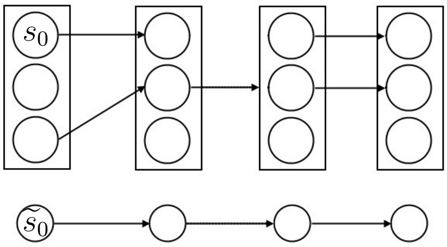

Figure 2.3: The generated counterexample may not be valid on the concrete one. In the abstract model, $\widetilde{s}_0$ is the abstract initial state that can reach to the abstract goal state through a path. As demonstrated in the figure, there is no path in the concrete model that begins with the initial state s_0.

in the model, or *loop* when a liveness property fails. Let S_0 be the set of initial states in the concrete model, $\widetilde{s}_0$ be the initial state in counterexample $\widetilde{ce}$ and h be the abstraction function. The counterexample identification algorithm (**SplitPATH**) [30] begins from the states of $st_0 = h^{-1}(\widetilde{s}_0) \cap S_0$. For all $0 < i < n$ where $n-1$ is the number of states over $\widetilde{ce}$, st_i contains all the successor states (or images) of st_{i-1} that fall into the set $h^{-1}(\widetilde{s}_i)$. It is proved that $\widetilde{ce}$ corresponds to a concrete counterexample if for all $0 < i < n$ we have $st_i \neq \emptyset$. In the case that i is the smallest index where $st_i = \emptyset$, then the counterexample $\widetilde{ce}$ is spurious and the state $\widetilde{s}_{i-1}$ is called *failure state*. If $\widetilde{s}_{i-1}$ is the failure state, then *dead-end states* are the *reachable states* in st_{i-1} with no outgoing transition to another state and *bad states* are unreachable states with outgoing transition to some states in st_i. The states that are not dead-end states or bad states are called *irrelevant states*. To prevent the spurious counterexample to occur again, we separate the set of dead-end states and bad states by splitting st_{i-1}. Clarke et al. proved that finding the *coarsest refinement* which separates the sets into the smallest size is NP-hard.

To identify spurious loop counterexamples, the loops will be *unwinded* and turned into a finite path. In general, unwinding algorithm may become exponential time, but the paper shows that only a polynomial time process is sufficient for spurious counterexample identification. The algorithm **SplitLOOP** in the paper finds the appropriate index of the failure state in the original counterexample.

CEGAR is used as the basic framework for many abstraction refinement techniques for software and hardware verification [91, 67] and have been used in various model-checking tools like SLAM [9, 8], BLAST [53, 54] and MAGIC [25]. The basic CEGAR framework is restricted to finite or infinite path counterexamples. In 2002, Clarke et al. [32] proposed a method for the generation of tree-like counterexamples for ACTL in such a way that they can be easily handled in the process of abstraction refinement.

2.4.2 Abstraction in model-checking multi-agent systems

In the recent years, several attempts have been made in the field of abstraction for multi-agent systems [44, 40]. Cohen et al. [33] first adopted the existential abstraction for interpreted systems. The specification is expressed in ACTLK, which is the ACTL logic with knowledge modality K. They modified the simulation relation defined in [30] for interpreted systems with respect to the *epistemic possibility relation* between the local states. It is proved in the paper that if model $\widetilde{I}$ simulates I, φ is an ACTLK formula and φ holds in $\widetilde{I}$, then φ holds in I. In their paper, they have shown that a *quotient* of an interpreted system which maps the local states, actions, local transitions and protocols into the equivalent classes will simulates the original one.

The paper has the importance of introducing the concept of existential abstraction into the multi-agent framework of interpreted systems. On the other hand, the abstraction mechanism is not automated. Moreover, no refinement methods is proposed in the case the abstract model does not satisfy the property.

2.4.3 Abstraction refinement for multi-agent systems

Maybe the first attempt to overcome the difficulties of abstraction refinement for the verification of epistemic properties is the recent research done by Zhou et al. [96]. They modified the tree-like counterexample generation method in [32] to cover the ACTLK specifications. The paper adopts similar approach as in [33] to build up an abstract model in which overestimates the concrete one. The main difference is that Cohen et al. approach is to abstract the model by first abstracting the agent-specific components like local states and local transitions, and then building up the abstract interpreted system. In this paper of Zhou et al. the global states will be split into equivalent classes and the global transition relation will get existentially quantified.

The main contribution of the paper more than abstraction and refinement is proposing a method for tree-like counterexample generation for ACTLK. A counterexample generated by verifying the specification φ is defined as an interpreted system in which (1) satisfies $\neg\varphi$ (2) underestimates the concrete model or in the other words, the concrete model simulates the counterexample. To find the counterexample, the mode-checker first builds the parse tree of the formula $\neg\varphi$ and traverses the tree in a depth-first manner. This process is not simple in the case when $\neg K_i$ appears in the formula. The authors proposed the procedure $\texttt{print\_witness}\bar{\texttt{K}}_i$ to output the counterexample related to the epistemic property. The paper have several major issues like (1) the authors consider a system with single initial state by assuming the possibility of transforming every interpreted system to the one with a single initial state (2) the refinement approach is not well explained especially when the counterexample is the result of verifying an epistemic property (3) case studies does not properly shows how the states are partitioned, what are the valuation of the propositions in each partition, and how the process of refinement proceeds.

In this thesis, we demonstrate a more detailed approach for the abstraction refinement for ACTLK properties.

CHAPTER 3

PRELIMINARIES

This chapter provides a policy language definition together with the materials and terminologies that are used in chapters 4 and 5 for the verification of dynamic access control systems.

3.1 Introduction

In a multi-agent system, the agents authenticate themselves by using the provided authentication mechanisms, such as login by username and password, and it is assumed that the mechanism is secure and reliable. Each agent is authorized to perform actions, which can change the system state by changing the values of several system variables (in our case, atomic propositions). Performing actions in the system encapsulates three aspects: the agent request for the action, allowance by the system and system transition to another state. In this thesis, we consider agents performing different actions *asynchronously*; a realistic approach in computer systems.

Asynchronous system: In *synchronous systems* agents can perform actions in parallel in each clock cycle, whereas in an *asynchronous systems* only one of the agents performs an action per clock cycle. One of the common problems in synchronous systems is the *race condition*. Let us demonstrate this problem with an example:

Example 3.1. Imagine a conference paper review system and two agents `Alice` and `Bob` as the reviewers of paper `p`. Consider the case where both the agents decide to assign `Tom` as the sub-reviewer of paper `p`, which is not a reviewer or sub-reviewer of `p`. Further assume that the security policy of the system contains the following rule:

- An agent can be assigned as the sub-reviewer of a paper if he is not already a reviewer or sub-reviewer of that paper.

If both `Alice` and `Bob` assign `Tom` as their sub-reviewer at the same time (same clock cycle), the precondition for the assignment is satisfied for both the reviewers. But after the assignment, `Tom` is assigned to the same paper by two reviewers, which is an unwanted situation. This problem does not occur when actions are performed asynchronously.

In our approach, we are *not* interested in security breaches caused by race condition. We consider that such issues are handled by memory locks or other application level methods. In general and in real systems, different requests are held in a queue and processed one at a time asynchronously. So, it is a realistic approach to model access control systems in asynchronous manner.

3.2 Access control policy

We present a simple policy language that is expressive enough to handle integrity constraints which are the rules that must remain true to preserve integrity of data, and policy invariants.

Syntax definition: Let T be a set of *types* which includes a special type `Agent` for agents and *Pred* be a finite set of *predicates* such that each n-ary predicate has a type $t_1 \times \cdots \times t_n \to \{\top, \bot\}$, for some $t_i \in T$, $1 \leq i \leq n$. Let V be a finite set of typed variables where the types are from the set T. We use the notation $\vec{v}$ to specify a sequence of distinct variables. An *atomic formula* is a predicate that is applied to a sequence of variables with the appropriate length and type.

The syntax of access control policy language is as follows:

$$\begin{array}{rrl}
 & L ::= & \top \mid \bot \mid w(\vec{v}) \mid L \vee L \mid L \wedge L \mid L \to L \mid \neg L \mid \forall v : t\ [L] \mid \exists v : t\ [L] \\
 & W ::= & +w(\vec{v}) \mid\ -w(\vec{v}) \mid \forall v : t.\ W \\
 & W_s ::= & W \mid W_s, W \\
\text{Action rule} & A_R ::= & \mathtt{id}(\vec{v}) : \{W_s\} \leftarrow L \\
\text{Read rule} & R_R ::= & \mathtt{id}(\vec{v}) : w(\vec{v}) \leftarrow L
\end{array}$$

In the above syntax, L is a *logical formula* and consists of atomic formulas combined by logical connectives and existential and universal quantifiers, $w \in \mathit{Pred}$, and $w(\vec{v})$ is an atomic formula. The formula L defines the condition for performing an action or reading an atomic formula. $\{W_s\}$ is the effect of the action rule that includes the updates. $+w(\vec{v})$ in the effect means executing the action will set the value of $w(\vec{v})$ to `true` and $-w(\vec{v})$ means setting the value to `false`. In the case of $\forall v.W$ in the effect, the action updates the signed atomic formula in W for all possible values of v. In the case that an atomic formula appears with different signs in multiple quantifications in the effect (for instance, $w(c,d)$ in $\forall x. + w(c,x), \forall y. - w(y,d)$), then only the sign of the last quantification is considered. The notation `id` indicates the rule identifier.

Let $a(\vec{v}) : E \leftarrow L$ be an action rule. The *free variables* of the logical formula L are denoted by $\mathbf{fv}(L)$ and are defined in the standard way. We also define $\mathbf{fv}(E) = \bigcup_{e \in E} \mathbf{fv}(e)$ where $\mathbf{fv}(\pm w(\vec{x})) = \vec{x}$ and $\mathbf{fv}(\forall x.W)$=$\mathbf{fv}(W) \backslash x$. We stipulate: $\mathbf{fv}(E) \cup \mathbf{fv}(L) \subseteq \vec{v}$. If $r(\vec{v}) : w(\vec{u}) \leftarrow L$ is a read rule, then $\vec{u} \cup \mathbf{fv}(L) \subseteq \vec{v}$.

In an asynchronous multi-agent system, it is crucial to know the agent that performs an action. Multi-agent system are any collection of interacting agents [46]. By definition, the first argument of an action rule is the agent that performs the action. The first argument of a read permission rule is the agent that reads the atomic formula to the left of the arrow.

Example 3.2. A conference paper review system policy contains the following properties for unassigning a reviewer from a paper:

- A chair is permitted to unassign the reviewers (`rev`).
- If a reviewer is removed, all the corresponding subreviewers (`subRev`) should be removed from the system at the same time.

The unassignment action rule can be formalized as follows:

$$\texttt{delRev}(u,p,a) : \{\texttt{-rev}(p,a), \forall b : \texttt{Agent}.\ \texttt{-subRev}(p,a,b)\} \leftarrow \texttt{chair}(u) \wedge \texttt{rev}(p,a)$$

Example 3.2 shows how updating several variables synchronously can preserve integrity constraints. The RW framework is unable to handle such integrity constraint as it can only update one proposition at a time.

3.3 Policy rule instantiation

Let Σ be a finite set of objects such that each object in Σ has a type. $\Sigma_t \subseteq \Sigma$ is the set of objects of type t. An atomic formula is *ground* if it is variable-free; i.e. its variables are substituted with the objects of the same type in Σ. For instance, if `reviewer`$\in$ *Pred* with two arities of type `Agent`, and `Bob`,`Paper`$\in \Sigma_{\texttt{Agent}}$, then `reviewer(Bob,Paper)` is a ground atomic formula. In the context of this thesis, we call the ground atomic formulas (atomic) *propositions*, since they only evaluate to `true` and `false`.

An *action* $\alpha : \varepsilon \leftarrow \ell$ contains an identifier α together with the *evolution rule* $\varepsilon \leftarrow \ell$, which is constructed by instantiating all the arguments in an action rule $a(\vec{v}) : E \leftarrow L$

with the objects of the same type in Σ. We refer to the whole action by its identifier α. Since the number of objects is finite, each quantified logical formula in L will be expanded to a finite number of conjunctions (for $\forall$ quantifier) or disjunctions (for $\exists$ quantifier) of logical formulas during the instantiation phase. The ground formula ℓ, which is the instantiation of L, determines the condition in which action α can be performed and is called *permission.* During the instantiation, the universal quantifiers in the effect will be expanded into a finite number of signed atomic propositions. If after instantiation, an atomic formula appears in the effect with different signs, we only consider the sign of the last occurrence.
For instance, if the policy contains the rule:

$$\texttt{assignReviewer}(x, y, p) : \{+\texttt{reviewer}(y, p)\} \leftarrow \texttt{chair}(x) \wedge \texttt{pcMember}(y) \wedge \neg\texttt{author}(p, y)$$

and `Alice, Bob` $\in \Sigma_{\texttt{Agent}}$, $\texttt{Paper}_1 \in \Sigma_{\texttt{Paper}}$, then the following instantiation of the rule:

$$\texttt{assignReviewer(Alice,Bob,Paper}_1\texttt{)} : \{+\texttt{reviewer(Bob,Paper}_1\texttt{)}\}$$
$$\leftarrow \texttt{chair(Alice)} \wedge \texttt{pcMember(Bob)} \wedge \neg\texttt{author(Paper}_1\texttt{,Bob)}$$

denotes an action where `Alice` assigns `Bob` as the reviewer of $\texttt{Paper}_1$. The right hand side of the arrow is a ground formula produced by substituting the variables x, y and p which is permission. If the permission is satisfied, then performing the action makes the system *evolve* by setting the value of $\texttt{reviewer(Bob,Paper}_1\texttt{)}$ to `true`. During the evolution, the values of all the propositions except the ones that appear in the effect remain the same.

A *read permission* $\rho : p \leftarrow \ell$ is constructed by instantiating the arguments in read permission rule $r(\vec{v}) : w(\vec{u}) \leftarrow L$ with the objects of the same type in Σ. ρ is the identifier, p is a proposition and ℓ is the condition for reading p.

Definition 3.1 (Active agent)**.** If α is an action, then $\mathbf{Ag}(\alpha)$ denotes the agent that performs α. As previously stated, this agent is the first argument of an action. For instance:

$$\mathbf{Ag}(\texttt{assignReviewer(Alice,Bob,Paper}_1\texttt{)}) = \texttt{Alice}$$

For a read permission $\rho : p \leftarrow \ell$, $\mathbf{Ag}(\rho)$ denotes the agent that reads the proposition p.

Definition 3.2 (Policy)**.** An *access control policy* denoted by $\mathcal{C}$ is a finite set of actions and read permissions derived by instantiating a set of rules with a finite set of typed objects.

Definition 3.3 (Action effect)**.** Let $\alpha : \varepsilon \leftarrow \ell$ be an action. Then we define:

$$\begin{aligned}\textbf{effect}_+(\alpha) &= \{p \mid +p \in \varepsilon\}\\ \textbf{effect}_-(\alpha) &= \{p \mid -p \in \varepsilon\}\\ \textbf{effect}(\alpha) &= \textbf{effect}_+(\alpha) \cup \textbf{effect}_-(\alpha)\end{aligned}$$

Example 3.3. In the action rule presented in example 3.2, assume we have 3 objects as $\Sigma_{\texttt{Agent}} = \{\mathsf{a}_1, \mathsf{a}_2\}$ and $\Sigma_{\texttt{paper}} = \{\mathtt{p}_1\}$. During instantiation phase, the action rule will be compiled into four instances: $\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_1)$, $\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_2)$, $\texttt{delRev}(\mathsf{a}_2, \mathtt{p}_1, \mathsf{a}_1)$ and $\texttt{delRev}(\mathsf{a}_2, \mathtt{p}_1, \mathsf{a}_2)$. The effect of the action $\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_2)$ will be:

$$\textbf{effect}_+(\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_2)) = \{\}$$
$$\textbf{effect}_-(\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_2)) = \{\texttt{rev}(\mathtt{p}_1, \mathsf{a}_2), \texttt{subRev}(\mathtt{p}_1, \mathsf{a}_2, \mathsf{a}_1), \texttt{subRev}(\mathtt{p}_1, \mathsf{a}_2, \mathsf{a}_2)\}$$
$$\textbf{effect}(\texttt{delRev}(\mathsf{a}_1, \mathtt{p}_1, \mathsf{a}_2)) = \{\texttt{rev}(\mathtt{p}_1, \mathsf{a}_2), \texttt{subRev}(\mathtt{p}_1, \mathsf{a}_2, \mathsf{a}_1), \texttt{subRev}(\mathtt{p}_1, \mathsf{a}_2, \mathsf{a}_2)\}$$

3.4 Summary

In this chapter we first introduced a policy language and then described the process of generating a set of actions and read permissions called policy, given a set of rules and a finite set of objects. In chapters 4 and 5, we use the same policy language for the description of access rules in order to build an access control model, but we evaluate the properties of different types.

CHAPTER 4

POLIVER: A KNOWLEDGE-BASED ACCESS CONTROL VERIFICATION TOOL

In this chapter, a new approach for automated knowledge-based verification of dynamic access control policies is presented. The verification method not only discovers if a vulnerability exists, but also produces the strategies that can be used by the attacker to exploit the vulnerability. It investigates the information needed by the attacker to achieve the goal and whether he acquires that information when he proceeds through the strategy or not. The algorithm is implemented and released as an open source policy verification tool called *PoliVer*.

The knowledge verification in PoliVer is limited to knowledge by readability. This abstraction of knowledge enhances the verification speed and memory usage of the tool. We argue that most - but not all - of the vulnerabilities can be investigated by this simplified concept of knowledge. In the next chapter, we will extend the knowledge-based verification to the knowledge by reasoning and present some vulnerabilities that can not be discovered by PoliVer and other access control verification tools.

Given the policy language in chapter 3, in this chapter we provide a verification algorithm which is able to find a strategy in a more efficient way than the guessing approach in a similar knowledge-based verification framework called RW [94]. This is because unlike RW which the verification algorithm is build around knowledge states and supports memoryful knowledge, the knowledge in our algorithm is memoryless and it is build around system states. But as our policy language supports variable bulk update, knowledge variables can be easily incorporated into the policy when it is required.

The rest of this chapter is organized as follows. The transition system and query language are introduced in Section 4.1. Model-checking strategy is explained in Section 4.2. Knowledge evaluation of the strategies is presented in Section 4.3. Experimental results are provided in section 4.5 and conclusions and future work are explained in Section 4.6.

4.1 Definitions

In this section, we describe how to build a labelled transition system from a policy. We also present a query language, which specifies the properties that we aim to verify over the system.

4.1.1 Building a labelled transition system from a policy

Given an access control policy (see definition 3.2), we build a labelled transition system as in the following definition.

Definition 4.1. Let $\mathcal{C}$ be a policy. Then the labelled transition system derived from $\mathcal{C}$ is:

$$M_{\mathcal{C}} = \langle S, Act, S_0, P, \tau, \gamma \rangle$$

where (1) P is the set of atomic propositions that appear in $\mathcal{C}$ (2) S is the set of states where each state is a valuation of the propositions in P (3) Act is the set of actions in $\mathcal{C}$ (4) $S_0 \subseteq S$ is the set of initial states (5) $\gamma : S \times P \to \{\top, \bot\}$ is the labelling function (6) $\tau : Act \times S \to S$ is the *partial transition function.* If $\alpha : \varepsilon \leftarrow \ell \in \mathcal{C}$ and ℓ holds in s, then $\tau(\alpha, s)$ is defined as s' such that

$$\gamma(s', p) = \begin{cases} \top & \text{if } +p \in \varepsilon \\ \bot & \text{if } -p \in \varepsilon \\ \gamma(s, p) & \text{Otherwise} \end{cases}$$

For the rest of this chapter, we use the shorthand notation $s \xrightarrow{\alpha} s'$ for stating $\tau(\alpha, s) = s'$. Note that as the set of initial states is not determined by the policy, the number of transition systems derivable from $\mathcal{C}$ is 2^n where $n = |S|$. Also read permissions in $\mathcal{C}$ are not involved in building the labelled transition system and will be considered in query evaluation and knowledge verification over the system.

4.1.2 Query language

Verification of the policy deals with the *reachability problem*, one of the most common properties arising in temporal logic verification. A state s is *reachable* if it can be reached in a finite number of transitions from the initial states. In multi-agent access control systems, the transitions are made by the agents performing actions.

The query language determines the *initial condition* and the *specification.* The syntax of the *policy query* is:

$$
\begin{aligned}
L ::= &\ \top \mid \bot \mid w(\vec{v}) \mid \langle w(\vec{v}) \rangle \mid L \vee L \mid L \wedge L \mid L \rightarrow L \mid \neg L \mid \forall v : t\ [L] \mid \exists v : t\ [L] \\
W ::= &\ w(\vec{v}) \mid w(\vec{v})* \mid w(\vec{v})! \mid w(\vec{v})*! \mid \neg W \\
W_s ::= &\ \text{null} \mid W_s, W \\
G ::= &\ C : (L) \mid C : (L \ \texttt{THEN}\ G) \\
\mathit{Query} ::= &\ \{W_s\} \rightarrow G
\end{aligned}
$$

where $w(\vec{v})$ is an atomic formula and C is a set of variables of type Agent.

In the above definition, G is called a nested goal if it contains the keyword THEN, otherwise it is called a simple goal. C is a *coalition of agents* interacting together to achieve the goal in the system. Also the agents in a coalition share the knowledge gained by reading system propositions or performing actions. The specification $\langle w(\vec{v}) \rangle$ means $w(\vec{v})$ is readable by at least one of the agents in the coalition. The initial condition is specified by the literals in $\{W_s\}$. Every literal W is optionally tagged with * when the value of atomic formula is fixed during verification, and/or tagged with ! when the value is initially known by at least one of the agents in the outermost coalition.

Example 4.1. One of the properties for a proper conference paper review system policy is that the reviewers (rev) of a paper should not be able to read other submitted reviews (submittedR) before they submit their own reviews. Consider the following query:

$$
\begin{aligned}
&\{\texttt{chair}(c)*!, \neg\texttt{author}(p,a)*, \texttt{submittedR}(p,b), \texttt{rev}(p,a), \neg\texttt{submittedR}(p,a)\} \rightarrow \\
&\{a\} : (\langle \texttt{review}(p,b) \rangle \wedge \neg\texttt{submittedR}(p,a)\ \ \texttt{THEN}\ \ \{a,c\} : (\texttt{submittedR}(p,a)))
\end{aligned}
$$

The query says "starting from the states satisfying the initial condition, is there any reachable state that agent a can promote himself in such a way that he will be able to read the review of the agent b for paper p while he has not submitted his own review and after that, agent a and c collaborate together so that agent a can submit his review of paper p?". If the specification holds, then there exists a security hole in the policy and should be fixed by policy designers. In the above query, the values of $\texttt{chair}(c)$ and $\texttt{author}(p,a)$ are fixed and $\texttt{chair}(c)$ is known to be true by the agent a at the beginning.

The above query is similar to the query 6.3 in RW [94] except the fact that in RW, the readability of $\texttt{review}(p,b)$ is memoryful (coalition memorizes the value of $\texttt{review}(p,b)$ whenever one of the agents reads its value), while in our case is memoryless.

Instantiation of the policy query: An *instantiated query* or simply *query* is the policy query with the variables substituted with the objects of appropriate type.

Definition 4.2 (Satisfaction relation)**.** Let $\mathcal{C}$ be a ground policy, *init* $\rightarrow g$ a query, C a coalition of agents, and $M_{\mathcal{C}}$ a derived transition system from $\mathcal{C}$ with the set of initial states S_0 as defined by *init*. Let **Ag** be the function defined by the definition 3.1. For any goal g, the notation $(M_{\mathcal{C}}, s, C) \models g$ means that given the coalition C, g holds in state s of the model $M_{\mathcal{C}}$. The satisfaction relation $\models$ is defined inductively as follows:

$(M_{\mathcal{C}}, s, C) \models p \quad \Leftrightarrow \quad \gamma(s, p) = \top$

$(M_{\mathcal{C}}, s, C) \models \neg\phi \quad \Leftrightarrow \quad (M_{\mathcal{C}}, s, C) \not\models \phi$

$(M_{\mathcal{C}}, s, C) \models \phi_1 \vee \phi_2 \quad \Leftrightarrow \quad (M_{\mathcal{C}}, s, C) \models \phi_1$ or $(M_{\mathcal{C}}, s, C) \models \phi_2$

$(M_{\mathcal{C}}, s, C) \models \langle p \rangle \quad \Leftrightarrow$ there exists a read permission $\rho : p \leftarrow \ell \in \mathcal{C}$ such that $\mathbf{Ag}(\rho) \in C$ and $(M_{\mathcal{C}}, s, C) \models \ell$

$(M_{\mathcal{C}}, s, C) \models C' : (\phi) \quad \Leftrightarrow$ there exists a path $s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $s = s_1$ and

(1) For all $1 \leq i < n$: $\mathbf{Ag}(\alpha_i) \in C'$

(2) For all $1 \leq i \leq n$: $(M_{\mathcal{C}}, s_i, C') \models p$ if $p* \in$ *init* and $(M_{\mathcal{C}}, s_i, C') \not\models p$ if $\neg p* \in$ *init*

(3) $(M_{\mathcal{C}}, s_n, C') \models \phi$

$(M_{\mathcal{C}}, s, C) \models C' : (\phi_1$ `THEN` $\phi_2) \quad \Leftrightarrow$ there exists a path $s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $s = s_1$ and

(1) For all $1 \leq i < n$: $\mathbf{Ag}(\alpha_i) \in C'$

(2) For all $1 \leq i \leq n$: $(M_{\mathcal{C}}, s_i, C') \models p$ if $p* \in$ *init* and $(M_{\mathcal{C}}, s_i, C') \not\models p$ if $\neg p* \in$ *init*

(3) $(M_{\mathcal{C}}, s_n, C') \models \phi_1$ and $(M_{\mathcal{C}}, s_n, C') \models \phi_2$

We use the notation $M_{\mathcal{C}} \models g$ if for all $s_0 \in S_0 : \; (M_{\mathcal{C}}, s_0, \emptyset) \models g$.

If a query is found to be positive in an access control system, then there exists a conditional sequence of actions called *strategy* (defined below) that makes the agents in the coalitions achieve the goal beginning from all the initial states. The strategy is presented formally by the following syntax:

$$\textit{strategy} ::= \text{null} \mid \alpha; \textit{strategy} \mid \textsf{if}(p) \; \{\textit{strategy}\} \; \textsf{else} \; \{\textit{strategy}\}$$

In the above syntax, p is an atomic proposition and α is an action. The value of p is not defined in the initial condition and can be true in some initial states and false in the others. If a strategy contains a condition over the proposition p, it means the value of p determines the required sequence of actions to achieve the goal. p is known as an *effective proposition* in our methodology.

Definition 4.3. (Transition relation). Given a labelled transition system M, let $s_1, s_2 \in S$ where S is the set of states, and ξ be a strategy. We use $s_1 \rightarrow_{\xi} s_2$ to denote "strategy ξ can be run in state s_1 and result in s_2", which is defined inductively as follows:

- $s \rightarrow_{\text{null}} s$.
- $s \rightarrow_{\alpha;\xi_1} s'$ if
  - If ℓ is the permission of α, then $(M, s, \emptyset) \models \ell$, and
  - $s'' \rightarrow_{\xi_1} s'$ where $s \xrightarrow{\alpha} s''$.
- $s \rightarrow_{\text{if}(p)\{\xi_1\} \text{ else } \{\xi_2\}} s'$ if:
  - If $\gamma(s, p) = \top$ then $s \rightarrow_{\xi_1} s'$ else $s \rightarrow_{\xi_2} s'$.

A set of states st_2 is *reachable* from the set of states st_1 through strategy ξ ($st_1 \rightarrow_{\xi} st_2$) if st_1 contains all the states s_1 which there exists $s_2 \in st_2$ such that $s_1 \rightarrow_{\xi} s_2$.

Definition 4.4. (State formula). If S is the set of states in labelled transition system M and $st \subseteq S$ then:

- f_{st} is a propositional formula satisfying exactly the states in st:
  $s \in st \;\leftrightarrow\; (M, s, \emptyset) \models f_{st}$
- st_f is the set of states satisfying f: $s \in st_f \;\leftrightarrow\; (M, s, \emptyset) \models f$

4.2 Model-checking and strategy synthesis

Our method uses backward search to find a strategy. Given a query $init \rightarrow g$ and model M with the initial states as defined by *init*, let *goal states* st_g be the set of states that satisfy the property of the innermost goal in g, and initials states be the states defined by *init*. The algorithm begins from st_g and finds all the states with transition to the current state, called pre-states. The algorithm continues finding pre-states over all found states until it gets all the initial states (success) or no new state could be found (fail).

The model-checking problem in this research is not a simple reachability question. As illustrated in figure 4.1, the strategy is successful only if it works for all the outcomes of reading or guessing a proposition in the model. Thus, reading/guessing behaviour produces the need for a universal quantifier, while actions are existentially quantified. The resulting requirement has an alternation of universal and existential quantifiers of arbitrary length, and this cannot be expressed using standard temporal logics such as CTL, LTL or ATL.

Notation 4.1. Assume f is a propositional formula. Then $p \in \textbf{prop}(f)$ if proposition p occurs in all formulas equivalent to f.

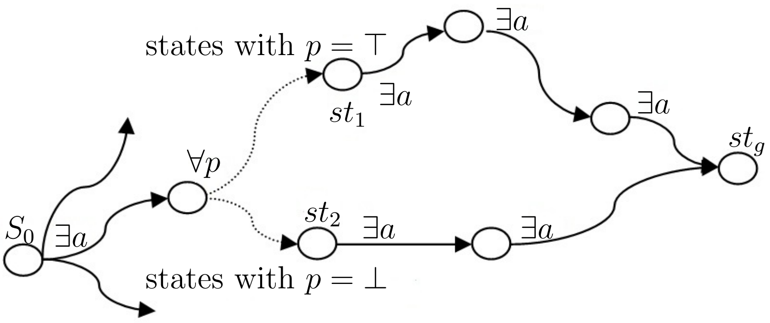

Figure 4.1: Strategy finding method. Ovals represent sets of states. Solid lines show the existence of an action that makes a transition between two sets of states. Dashed lines are universally quantified over the outcome of reading or guessing the value of proposition p.

Definition 4.5. (Pre states). Let $init \to g$ be a query, $M_{\mathcal{C}}$ be a labelled transition system derived from policy $\mathcal{C}$ with the initial states as defined by $init$. If action $\alpha : \varepsilon \leftarrow \ell \in Act$ and $st \subseteq S$, then $PRE^{\exists}_{\alpha}(st)$ is the set of states in which action α is permitted to perform and performing the action will make a transition to one of the states in st by changing the values of the propositions in the effect of the action. Let Lit^{*} be the set of literals that are tagged by $*$ in $init$. Then:

$$PRE^{\exists}_{\alpha}(st) = \Big\{ s \in S \mid (M, s, \emptyset) \models \ell, \tau(\alpha, s) = s', s' \in st \text{ and for all } l \in Lit* : (M, s, \emptyset) \models l \Big\}$$

The symbolic (BDD-based) presentation of $PRE^{\exists}_{\alpha}$ is contained in appendix A.

4.2.1 Finding effective propositions

Definition 4.6. (Effective proposition). In labelled transition system M, atomic proposition p is *effective* with respect to S_0 as the set of initial states and st_g as the set of goal states if there exists a set of states $st \subseteq S$ and strategies ξ_0, ξ_1 and ξ_2 such that $\xi_1 \neq \xi_2$ and:

- $S_0 \to_{\xi_0} st$,
- $st \cap \{s \mid \gamma(s,p) = \top\} \to_{\xi_1} st_g$,
- $st \cap \{s \mid \gamma(s,p) = \bot\} \to_{\xi_2} st_g$ and
- $st \cap \{s \mid \gamma(s,p) = \top\} \neq \emptyset,\ st \cap \{s \mid \gamma(s,p) = \bot\} \neq \emptyset$.

Effective propositions are important for the following reason:

The value of proposition p is not specified in the query and is not known by the agents at the beginning. The agents need to know the value of p to select the appropriate strategy to achieve the goal. In the states of st, if the agent (or coalition of agents) knows the value of p, he will perform the next action without taking any risk. Otherwise, he needs to guess the value of p. This situation is risky and in the case of a wrong decision and may not be repeatable.

The algorithm provided in this section is capable of finding effective propositions while searching for strategies, and then, is able to verify the knowledge of the agents about effective propositions in the decision states. The algorithm is guided by proposition 4.5 in order to detect effective propositions in backward search.

Proposition 4.1. Suppose that f is a well-formed first order logic formula, C is a coalition of agents, p is a proposition and s is a state in labelled transition system M. Suppose that $(M, s, C) \models p$ and $(M, s, C) \models f[\top/p]$. Then we have $(M, s, C) \models f$.

Proof. The proof proceeds by structural induction. Assume that f is in NNF (negation normal form).

Base cases:

- $f = p$: By hypothesis, $(M, s, C) \models p$ and $(M, s, C) \models \top$. Then $(M, s, C) \models p$.
- $f = q$, $q \neq p$: By hypothesis, $(M, s, C) \models p$ and $(M, s, C) \models q$. Then we have $(M, s, C) \models q$.
- $f = \neg p$: By hypothesis, $(M, s, C) \models p$ and $(M, s, C) \models \bot$. This case is impossible.
- $f = \neg q$, $q \neq p$: By hypothesis, $(M, s, C) \models p$ and $(M, s, C) \models \neg q$. Therefore $(M, s, C) \models \neg q$.

Inductive cases:

Assume by inductive hypothesis that for two given well-formed logical formulas f_1 and f_2 that are presented in NNF, if $(M, s, C) \models p$ and $(M, s, C) \models f_1[\top/p]$ then $(M, s, C) \models f_1$, and if $(M, s, C) \models p$ and $(M, s, C) \models f_2[\top/p]$ then $(M, s, C) \models f_2$.

Case 1: We need to show that if $(M, s, C) \models p$ and $(M, s, C) \models (f_1 \wedge f_2)[\top/p]$, then $(M, s, C) \models f_1 \wedge f_2$.

If $(M, s, C) \models (f_1 \wedge f_2)[\top/p]$ then $(M, s, C) \models f_1[\top/p]$ and $(M, s, C) \models f_2[\top/p]$ holds. By inductive hypothesis, we have $(M, s, C) \models f_1$ and $(M, s, C) \models f_2$, which is equivalent to $(M, s, C) \models f_1 \wedge f_2$.

Case 2: For the second inductive case, we need to show that if $(M, s, C) \models p$ and $(M, s, C) \models (f_1 \vee f_2)[\top/p]$, then $(M, s, C) \models f_1 \vee f_2$.

If $(M, s, C) \models (f_1 \vee f_2)[\top/p]$ then $(M, s, C) \models f_1[\top/p]$ or $(M, s, C) \models f_2[\top/p]$ holds. By inductive hypothesis, we have $(M, s, C) \models f_1$ or $(M, s, C) \models f_2$, which is equivalent to $(M, s, C) \models f_1 \vee f_2$.

So, we can conclude that for every well-formed formula f, proposition 4.1 holds. □

Proposition 4.2. Suppose that f is a well-formed first order logic formula, C is a coalition of agents, p is a proposition and s is a state in labelled transition system M. Suppose that $(M, s, C) \models \neg p$ and $(M, s, C) \models f[\bot/p]$. Then we can conclude that $(M, s, C) \models f$.

Proof. The proof is similar to the proof for the proposition 4.1. □

Proposition 4.3. Let st_1 be a set of states in labelled transition system M and $p \in \mathbf{prop}(f_{st_1})$. Suppose $s \in st_{f_{st_1[\top/p]}}$ and $\gamma(s, p) = \top$, then $s \in st_1$.

Proof. By definition 4.4, $s \in st_{f_{st_1[\top/p]}}$ is equivalent to $(M, s, C) \models f_{st_1}[\top/p]$. Also $\gamma(s, p) = \top$ is equivalent to $(M, s, C) \models p$. By proposition 4.1, $(M, s, C) \models f_{st_1}$, which allows us to conclude $s \in st_1$. □

Proposition 4.4. Let st_1 be a set of states in labelled transition system M and $p \in \mathbf{prop}(f_{st_1})$. Suppose $s \in st_{f_{st_1[\bot/p]}}$ and $\gamma(s, p) = \bot$, then $s \in st_1$.

Proof. The proof is similar to the proof for proposition 4.3. □

Proposition 4.5. Let st_1, st_2 and st_g be sets of states and ξ_1 and ξ_2 be strategies such that $st_1 \rightarrow_{\xi_1} st_g$ and $st_2 \rightarrow_{\xi_2} st_g$. Suppose $p \in \mathbf{prop}(f_{st_1}) \cap \mathbf{prop}(f_{st_2})$, $st_d = st_{f_{st_1[\top/p]}} \cap st_{f_{st_2[\bot/p]}}$ and $s \in st_d$. Then if $\gamma(s, p) = \top$, we conclude that $s \rightarrow_{\xi_1} st_g$, otherwise $s \rightarrow_{\xi_2} st_g$ will be concluded.

Proof. If $s \in st_d$ then $s \in st_{f_{st_1}[\top/p]}$. If $\gamma(s, p) = \top$ then by proposition 4.3, $s \in st_1$ and therefore $s \rightarrow_{\xi_1} st$. If $\gamma(s, p) = \bot$, since $s \in st_{f_{st_2}[\bot/p]}$, then by proposition 4.4 we have $s \in st_2$, resulting in $s \rightarrow_{\xi_2} st$. □

Let st_g in proposition 4.5 be the set of goal states, st_d the set of states found according to the proposition 4.5 and S_0 the set of initial states. If there exist a strategy ξ_0 such that $S_0 \rightarrow_{\xi_0} st_d$, then by definition 4.6, the atomic proposition p is an effective proposition and therefore $st_d \rightarrow_{\text{if}(p)\ \{\xi_1\}\ \text{else}\ \{\xi_2\}} st_g$. The states in st_d are called *decision states.*

Example 4.2. Let the following be the policy rules for changing a password in a system where the arguments of the predicates are of type `Agent`:

$$\texttt{setTrick}(a) : \{+\texttt{trick}(a)\} \leftarrow \neg\texttt{permission}(a),$$
$$\texttt{changePass}(a) : \{+\texttt{passChanged}(a)\} \leftarrow \texttt{permission}(a) \vee \texttt{trick}(a)$$

In the above policy rules, the administrator of the system has defined permission for changing password. The permission declares that one of the atomic formulas `permission`(a) or `trick`(a) is needed for changing password. `permission`(a) is write protected for the agents and no action is defined for changing it. If an agent does not have permission to change his password, he can set $\texttt{trick}(a)$ to true first and then, he will be able to change the password. This can be seen as a mistake in the policy. We have excluded read permission rules, as they are not required in this particular example for query verification and will be considered only in knowledge verification phase.

Consider that we have just one object of type Agent in the system ($\Sigma_{\texttt{Agent}} = \{\texttt{a}_1\}$) and we want to verify the query $\{\} \to \{a\} : (\texttt{passChanged}(a))$. The only possible instantiation of the query is when variable a is assigned to $\texttt{a}_1$. As the initial condition is empty, no condition is defined to specify the initial states and therefore they cover all the system states. Let policy $\mathcal{C}$ be derived by instantiating the rules with agent $\texttt{a}_1$ and $M_{\mathcal{C}}$ be the labelled transition system derived from $\mathcal{C}$ with S as the set of states where each state in S is a valuation of the propositions $\texttt{trick}(\texttt{a}_1)$, $\texttt{permission}(\texttt{a}_1)$ and $\texttt{changePass}(\texttt{a}_1)$, and $S_0 = S$. The following procedures show how a strategy can be found:

$$f_{st_g} = \texttt{passChanged}(\texttt{a}_1)$$

We can find one set of states as the pre-state of st_g:

$$f_{PRE^{\exists}_{\texttt{changePass}(\texttt{a}_1)}(st_g)} = f_{st_1} = \texttt{permission}(\texttt{a}_1) \vee \texttt{trick}(\texttt{a}_1)$$

$$st_1 \to_{\texttt{changePass}(\texttt{a}_1)} st_g$$

f_{st_g} and f_{st_1} don't share any proposition and hence, proposition 4.5 is not applicable. For the set st_1, we can find one pre-set:

$$f_{PRE^{\exists}_{\texttt{setTrick}(\texttt{a}_1)}(st_1)} = f_{st_2} = \neg\texttt{permission}(\texttt{a}_1)$$

$$st_2 \to_{\texttt{setTrick}(\texttt{a}_1);\texttt{changePass}(\texttt{a}_1)} st_g$$

Based on propositon 4.5 and for the states st_1 and st_2 and proposition $p = \texttt{permission}(\texttt{a}_1)$ we have:

$$f_{st_1}[\top/p] = \top, f_{st_2}[\bot/p] = \top, f_{st_1}[\top/p] \wedge f_{st_2}[\bot/p] = \top$$

$$st_3 = st_\top = S \qquad\qquad st_3 \to_\xi st_g$$

$$\xi = \textsf{if } (\texttt{permission}(\texttt{a}_1))\{\texttt{changePass}(\texttt{a}_1)\} \textsf{ else } \{\texttt{setTrick}(\texttt{a}_1); \texttt{changePass}(\texttt{a}_1)\}$$

Since $S_0 \subseteq st_3$, the goal is reachable and we output the strategy.

Backward search transition filtering: If an action changes a proposition, the value of the proposition will be set and known for the rest of the strategy. So in backward search algorithm, we filter out the transitions that alter effective propositions before their corresponding decision states are reached.

Definition 4.7. (State strategy). Consider st_g as the set of goal states. *State strategy* is the triple (st, ξ, efv) defined inductively as follows:

- $(st_g, \text{null}, \emptyset)$ is a state strategy.
- $(st, \alpha; \xi, \mathit{efv})$ is a state strategy if:
  - For all $p \in \mathbf{effect}(\alpha)$: $p \notin \mathit{efv}$ and
  - (st', ξ, efv) is a state strategy where $st' = \{s' \mid s \xrightarrow{\alpha} s', s \in st\}$.
- $(st, \mathsf{if}(p)\{\xi_1\}\ \mathsf{else}\ \{\xi_2\}, \mathit{efv})$ is a state strategy if:
  - $p \in \mathit{efv}$,
  - $(st_{f_{st} \wedge p}, \xi_1, \mathit{efv} \backslash \{p\})$ is a state strategy and
  - $(st_{f_{st} \wedge \neg p}, \xi_2, \mathit{efv} \backslash \{p\})$ is a state strategy.

In the above definition, *st* contains all the states that some states in st_g are reachable from them through the strategy ξ, and *efv* is the set of effective propositions in ξ. The definition enforces a control condition in the verification process, preventing effective propositions from being altered in previous steps.

4.2.2 Pseudocode for finding strategy

Let $\mathcal{C}$ be a policy, $\mathit{init} \to C(L)$ a simple query and M a derived labelled transition system from $\mathcal{C}$ with the set of initial states S_0 as defined by *init*. Let P be the set of atomic propositions, $A_C \in \mathit{Act}$ the set of all the actions that the agents in coalition C can perform, and st_g the set of all the states $s \in S$ where $(M, s, C) \models L$. K_C contains the propositions known by the agents in coalition C at the beginning (tagged with ! in *init*). The triple (st, ξ, efv) is the state strategy, which keeps the set of states *st* found during backward search, the strategy ξ to reach the goal from *st* and the set of effective propositions *efv* occurring in ξ. The pseudocode for the strategy finding algorithm is demonstrated in Algorithm 1.

In Algorithm 1, the outermost while loop checks the fixed point of the algorithm, where no more state (or equivalently, state strategy) could be found in backward search. Inside the while loop, the algorithm traverses the state strategy set that contains $(st_g, \text{null}, \emptyset)$ at the beginning. For each state strategy (st, ξ, efv), it finds all the possible pre-states for *st* and appends the corresponding state strategies to the set. It also finds effective propositions and decision states by performing pairwise analysis between all the members

Algorithm 1 Strategy finding algorithm

```
▷ **Input**: $S_0$ is the set of initial states, $st_g$ is the set of goal states, $P$ is the set of atomic propositions, $A_C$ is the set of actions the coalition $C$ can perform and $K_C$ is the set of known propositions by the coalition $C$.
▷ **Output**: returns a set of strategies.

$state\_strategies := \{(st_g, \text{null}, \emptyset)\}$
$states\_seen := \emptyset$
$old\_strategies := \emptyset$

while $old\_strategies \neq state\_strategies$ do
    $old\_strategies := state\_strategies$
    for all $(st_1, \xi_1, efv_1) \in state\_strategies$ do
        for all $\alpha \in A_C$ do
            if effect$(\alpha) \cap efv_1 = \emptyset$ then
                $PRE := PRE^{\exists}_{\alpha}(st_1)$
                if $PRE \neq \emptyset$ and $PRE \not\subseteq states\_seen$ then
                    $states\_seen := states\_seen \cup PRE$
                    $\xi :=$ “$\alpha$;” $+ \xi_1$
                    $state\_strategies := state\_strategies \cup \{(PRE, \xi, efv_1)\}$
                    if $S_0 \subseteq PRE$ then
                        output $\xi$
                    end if
                end if
            end if
        end for

        for all $(st_2, \xi_2, efv_2) \in state\_strategies$ do
            for all $p \in P \backslash K_C$ do
                if $p \in$ prop$(f_{st_1}) \cap$ prop$(f_{st_2})$ then
                    $PRE := st_{f_{st_1[\top/p]}} \cap st_{f_{st_2[\bot/p]}}$
                    if $PRE \neq \emptyset$ and $PRE \not\subseteq states\_seen$ then
                        $states\_seen := states\_seen \cup PRE$
                        $\xi :=$ “if($p$)” $+ \xi_1 +$ “else” $+ \xi_2$
                        $state\_strategies := state\_strategies \cup \{(PRE, \xi, efv_1 \cup efv_2 \cup \{p\})\}$
                        if $S_0 \subseteq PRE$ then
                            output $\xi$
                        end if
                    end if
                end if
            end for
        end for
    end for
end while
```

of the state strategy set based on the proposition 4.5. The strategy will be returned if the initial states are found in backward search.

Proposition 4.6. (Termination) The algorithm eventually terminates.

Proof. The algorithm terminates when the while-loop terminates. The loop terminates if: the inner for-loops terminate, and no new state strategy could be found. The for-loops in lines 11 and 26 iterate over fixed size sets and will eventually terminate. The for-loops in lines 10 and 25 iterate over the set *state_strategies* that may increase in size in each loop iteration. The loops add a new state strategy to the set if some states that were not seen before are encountered (lines 13 and 28). By the fact that the number of states are finite, the size of *state_strategies* is bounded by the maximum number of states and correspondingly, the number of for-loop iterations are bounded. By the same reason, we can conclude that the number of newly found states is also bounded by the maximum number of states and the while-loop terminating condition will eventually satisfy. □

Proposition 4.7. For all $(st, \xi, \mathit{efv}) \in \mathit{state\_strategies}$, st contains all the states s in which ξ can be run in s and result in $s' \in st_g$.

Proof. The proof proceeds by induction over the set *state_strategies*.
Base case:

- $\mathit{state\_strategies} = \{(st_g, \text{null}, \emptyset)\}$: The proposition trivially holds.

Inductive cases: We assume by inductive hypothesis that the proposition holds for all $(st, \xi, \mathit{efv}) \in \mathit{state\_strategies}$. Then two different state strategies may be added to the set in the next while loop iteration:

- $(PRE^{\exists}_{\alpha}(st), \alpha; \xi, \mathit{efv})$ where $(st, \xi, \mathit{efv}) \in \mathit{state\_strategies}$:

  The set $PRE^{\exists}_{\alpha}(st)$ contains all the states s where $s \xrightarrow{\alpha} s'$ and $s' \in st$. As st contains all the states that can reach the states in st_g though strategy ξ, therefore $PRE^{\exists}_{\alpha}(st)$ contains all the states that reach the states in st_g through $\alpha; \xi$. The effective propositions in the two strategies are the same.

- $(st', \text{if}(p)\{\xi_1\} \text{ else } \{\xi_2\}, \mathit{efv}_1 \cup \mathit{efv}_2 \cup \{p\})$ where

  - $(st_1, \xi_1, \mathit{efv}_1), (st_2, \xi_2, \mathit{efv}_2) \in \mathit{state\_strategies}$, and
  - $st' = st_{f_{st_1[\top/p]}} \cap st_{f_{st_2[\bot/p]}}$.

  Let $s \in st'$. If $\gamma(s, p) = \top$, then by proposition 4.5, ξ_1 can be run in s and result in a state in st_g. For the case of $\gamma(s, p) = \bot$, the same is true with the strategy ξ_2. So the proposition holds for all the states in st'.

□

Proposition 4.8. (Soundness) If the algorithm outputs a strategy, it can be run over S_0 and results in st_g.

Proof. The algorithm outputs a strategy whenever it finds a $(st, \xi, \mathit{efv}) \in \mathit{state\_strategies}$ such that $S_0 \subseteq st$. By proposition 4.7, st contains all the states (icluding S_0) which ξ can be run on them and result in st_g. Therefore, the proposition holds. □

The following Lemma will be used in order to proof the completeness of the algorithm.

Lemma 4.1. Suppose Algorithm 1 is run with input st_g and terminates with a value for *state_strategies*. Let $st_0 \subseteq S$. If $st_0 \rightarrow_\xi st_g$, then there exists $(PRE, \xi', \mathit{efv}) \in \mathit{state\_strategies}$ such that $st_0 \subseteq PRE$.

Proof. The proof proceeds by induction over height of ξ.

Base case: $\xi = \text{null}$ and therefore $st_0 = st_g$. By default, $(st_g, \text{null}, \emptyset) \in \mathit{state\_strategies}$ and therefore the statement trivially holds, with $PRE = st_g$, $\xi' = \text{null}$ and $\mathit{efv}= \emptyset$.

Inductive case: Assume by inductive hypothesis that the statement holds for all the strategies of height up to n. Let $st_0 \rightarrow_\xi st_g$ where ξ is a strategy of length $n + 1$. Then

- $\xi = \alpha; \xi_1$ where ξ_1 is of height n. So we have $st_0 \rightarrow_\alpha st_1 \rightarrow_{\xi_1} st_g$. By hypothesis, there exists $(PRE_1, \xi'_1, \mathit{efv}_1) \in \mathit{state\_strategies}$ such that $st_1 \subseteq PRE_1$. By definition 4.7 and $st_0 \rightarrow_\alpha st_1 \rightarrow_{\xi_1} st_n$, the condition on line 12 of the algorithm holds. Hence, the algorithm finds the set $PRE = PRE^{\exists}_{\alpha}(PRE_1)$ which is the set of all the states with a transition to some states in PRE_1 as the result of performing action α in them. Therefore we have $st_0 \subseteq PRE$. The set PRE is not empty as it already contains the states in st_0. If the states of PRE are met before (line 14), then the state strategy $(PRE, \alpha; \xi'_1, \mathit{efv}_1)$ will be added to the set *state_strategies* and the statement holds. Otherwise, such a state strategy already exists.

- $\xi = \text{if}(p)\{\xi_1\}\text{else}\{\xi_2\}$ where the maximum height of ξ_1 and ξ_2 is n. By definition 4.7, $st_{f_{st_0} \wedge p} \rightarrow_{\xi_1} st_g$ and $st_{f_{st_0} \wedge \neg p} \rightarrow_{\xi_2} st_g$. By hypothesis, there exists $(PRE_1, \xi'_1, \mathit{efv}_1), (PRE_2, \xi'_2, \mathit{efv}_2) \in \mathit{state\_strategies}$ such that $st_{f_{st_0} \wedge p} \subseteq PRE_1$ and $st_{f_{st_0} \wedge \neg p} \subseteq PRE_2$. As p is found to be an effective proposition in a strategy, $p \notin K_C$ and therefore the algorithm enters the loop on line 26 for p. Consider three cases for line 27: (1) $p \notin \textbf{prop}(PRE_1)$. Then $st_{f_{st_0} \wedge p} \subseteq st_{f_{st_0}} = st_0 \subseteq PRE_1$ and therefore the statement holds for the state strategy $(PRE_1, \xi'_1, \mathit{efv}_1)$ (2) $p \notin \textbf{prop}(PRE_2)$. Similar to the previous case, the statement holds for the state strategy $(PRE_2, \xi'_2, \mathit{efv}_2)$. (3) $p \in \textbf{prop}(PRE_1) \cap \textbf{prop}(PRE_2)$, Then we have $st_0 \subseteq PRE_1[\top/p] \cap PRE_2[\bot/p]$. Using the same arguments as the first item for the conditions on line 29, $(PRE_1[\top/p] \cap PRE_2[\bot/p], \text{if}(p)\{\xi_1'\}\text{else}\{\xi_2'\}, \mathit{efv}_1 \cup \mathit{efv}_2 \cup \{p\})$ will be added to the set *state_strategies* and the statement holds for that state strategy.

□

Proposition 4.9. (Completeness) If some strategy exists from S_0 to st_g, then the algorithm will find one.

Proof. If such strategy exists (let us say ξ), then by Lemma 4.1, the algorithm will find a state strategy $(PRE, \xi', \mathit{efv})$ where $S_0 \subseteq PRE$. In that case lines 19 and 34 of the algorithm guarantee that the strategy ξ' will be delivered. Note that ξ' may be different from ξ. □

Verification of the nested goals: To verify a nested goal, we begin from the inner-most goal. By backward search, all backward reachable states will be found and their intersection with the states for the outer goal will construct the new set of goal states. For the outer-most goal, we look for the initial states between backward reachable states. If we find them, we output the strategy. Otherwise, the nested goal is unreachable.

4.3 Knowledge vs. guessing in strategy

Agents in a coalition know the value of a proposition if: they have read the value before, or they have performed an action that has affected that proposition[1]. If a strategy is found, we are able to verify the knowledge of the agents over the strategy and specifically for effective propositions, using read permissions defined in the policy. Read permissions don't lead to any transition or action, and are used just to detect if an agent or coalition of agents can find out the way to the goal with complete or partial knowledge of the system. The knowledge is shared between the agents in a coalition.

To find agent knowledge over effective propositions, we begin from the initial states, run the strategy and verify the ability of the coalition to read the effective propositions. If at least one of the agents in the coalition can read an effective proposition before or at the corresponding decision states, then the coalition can find the path without taking any risk. In the lack of knowledge, agents should guess the value in order to find the next required action along the strategy.

Pseudocode for knowledge verification over the strategy: Let $\mathcal{C}$ be a policy, $\mathit{init} \rightarrow C(L)$ a simple query and $M_{\mathcal{C}}$ a derived labelled transition system from $\mathcal{C}$ with

[1] In this research, we do not consider reasoning about knowledge like the one in interpreted systems. This approach makes the concept of knowledge weaker, but more efficient to verify.

Procedure 2 Knowledge verification function

```
function KnowledgeAlgo($st, \xi, efv, C, k_C$)
    ▷ Input: $st$ is a set states, $\xi$ is a strategy, $efv$ is the set of effective propositions occurring in $\xi$, $C$ is the coalition of agents, and $k_C$ is the knowledge of the coalition.
    ▷ Output: returns the annotated strategy.

    if $\xi$=null then
        return null
    end if
    for all $p \in efv, u_1 \in$ C do
        for all read permissions $\rho(u_1, \vec{o}) : p \leftarrow \ell \in \mathcal{C}$ do
            if for all $s \in st : (M_{\mathcal{C}}, s, C) \models \ell$ then
                $k_C := k_C \cup \{p\}$
            end if
        end for
    end for

    if $\xi = \alpha; \xi_1$ then
        $st' := \{s' \mid s \xrightarrow{\alpha} s', s \in st\}$
        return "$\alpha$;"+ KnowledgeAlgo($st', \xi_1, efv, C, k_C \cup$ effect($\alpha$))
    else if $\xi =$ if($p$){$\xi_1$} else {$\xi_2$} then
        if $p \in k_C$ then
            $str$ :=""
        else
            $str$ :="Guess: "
        end if
        return $str$+ "if($p$){"+
            KnowledgeAlgo($st_{f_{st} \wedge p}, \xi_1, efv \backslash \{p\}, C, k_C$) + "}else{"+
            KnowledgeAlgo($st_{f_{st} \wedge \neg p}, \xi_2, efv \backslash \{p\}, C, k_C$)+"}"
    end if
end function
```

the set of initial states S_0 as defined by *init*. Let ξ be the strategy that found by the Algorithm 1 with the state strategy (S_0, ξ, efv). Therefore we have $S_0 \rightarrow_\xi st_g$ where st_g is the set of all the states $s \in S$ in which $(M_{\mathcal{C}}, s) \models L$. If K_C contains the set of propositions that are tagged with ! in *init* at the beginning, then the recursive function $\textsc{KnowledgeAlgo}(S_0, \xi, efv, C, K_C)$ returns an annotated strategy with a string Guess: added to the beginning of every if statement in ξ, where the coalition does not know the value of the proposition inside if statement.

Knowledge verification for nested goals: To handle knowledge verification over the strategies found by nested goal verification, we begin from the outermost goal. We traverse over the strategy until the goal states are reached. For the next goal, all the accumulated knowledge will be transferred to the new coalition if there exists at least one common agent between the two coalitions. The algorithm proceeds until the strategy is fully traversed.

4.4 Implementation and case studies

PoliVer is implemented by modifying the model-checker AcPeg [95, 94]. We have kept some useful syntactic and functional properties implemented in AcPeg like the structure of policy definition and query statement. In the implementation, the syntax of action rules and read permission rules presented in chapter 3 is modified in order to provide a more user friendly language. PoliVer is implemented in Java and can be run over different platforms. For symbolic model-checking, we have used BuDDy as a well-known binary decision diagram library.

4.4.1 PoliVer input script

An access control model in PoliVer script is composed of a policy, a run-statement and a query-statement.

policy definition

The policy begin with the keyword `AccessControlSystem` followed by an identifier as the name of the model. Type-definition is the first block of the policy which begins with the keyword `Class` followed by comma separated identifiers for the types. Type identifiers must begin with capital letter. Type-definition ends with a semicolon. The next block is predicate-definition. This block starts with the keyword `Predicate` followed by comma separated parametrized predicates. Parameter names begin with lower case letters, they must be distinct and their types should be declared. For example, in example 4.2 the predicate `permission` has only one argument of type `Agent`.

The next component of a policy is action-rule definitions. Action rules begin with the keyword `Action` and use the similar syntax as in chapter 3 with some modifications to make the policy definition user friendly:

- `Agent` is a predefined type and it is not required to be defined in the type definition block.
- The agent that performs the action is syntactically abstracted as the first parameter of the action rule. The keyword `user` is reserved to demonstrate the agent that performs the action in the body of the action rule.
- The operator "=" is used to define the equivalence of two objects of the same type. For example, `user` $= a$ where a is a parameter denotes that a is the same agent as `user`.

Each action rule ends with a semicolon. In PoliVer syntax, the operator $\wedge$ is replaced with & and `and`, $\vee$ is replaced with | and `or`, $\neg$ is replaced with ∼ and quantifiers $\forall$ and $\exists$ are substituted by `E` and `A` respectively.

The last component of a policy is read permission rules. Read permission rules start with the keyword `Read`. Again for simplicity, we have abstracted the agent who can read the truth value of the instanced predicate from the arguments. We have replaced that agent with the keyword `user` in the body of the rule. This abstraction leads to a more simplified form of read permission rules where the identifier is completely abstracted. For example:

```
Read reviewer(p, a) <- pcmember(user) & ∼author(p, user);
```

defines the condition in which the agent `user` is able to read `reviewer(p, a)`. Note that `user`, `p` and `a` are place holders and will be replaced with the appropriate objects during instantiation phase.

Policy definition terminates with the keyword `end`.

Run-statement

Instead of explicitly defining the set of objects with different types and declaring the objects in each set, we can simply let PoliVer automatically generate the objects and populate the policy with them to build the model. The run statement declares the number of objects for each type and is specified before the query statement. It begins with the keyword `run for` followed by comma separated pairs if numbers and types. For instance:

```
run for 2 Agents, 3 Papers
```

informs PoliVer to assign 2 elements to the set `Agents` and three to the set `Papers`. For the above run-statement, the predicate `reviewer(p:Papers, a:Agents)` will be instantiated into six system propositions.

Query-statement

The query-statement begins with the keyword `check` and is of the form `check` {L || Query} where L contains quantified variables that occur in Query, and Query is of the form defined in section 4.1.2. For example and for a query in a conference paper review system, L can be of the form `E disj a,c:Agent, p:Paper`. The existential quantification of (typed) variables in L informs PoliVer to check if the property holds for some instance of the variables `a`, `c` and `p`. In the case of universal quantification, PoliVer checks if the property holds for all possible instances of a quantified variable. The keyword `disj` is also used to notify PoliVer not to assign the same object to the variables in the scope of a quantifier (`a` and `c` in the example).

The syntax of the initial condition in the query is also remained similar to the one in AcPeg for the implementation. Therefore instead of a set of literals, we use a conjunction of the literals for the initial condition. For instance, the initial condition {`chair(c)*!`,¬`author(p,a)*!`} is transformed to `chair(c)*! &` ∼`author(p,a)*!` in the implementation.

4.4.2 Case studies

A conference paper review system (CRS) Assume a conference paper review system with the following rules:

1. A chair can assign an agent as a PC member and a PC member can resign his membership.

2. A chair can assign a PC member as a reviewer of a paper with the constraint that the reviewer should not be the author of his assigned papers.

3. A reviewer can resign as the reviewer of a paper. At the same time, all the sub-reviewers that he has appointed to that paper should get removed. A sub-reviewer can resign if he has not submitted his review.

4. A reviewer of a paper can allocate an agent to be a sub-reviewer of the paper if the agent is not the author and a sub-reviewer of that paper.

5. A reviewer or sub-reviewer of a paper can submit his review if he has not submitted the review before.

6. A reviewer or sub-reviewer of a paper can write or update his review before submission.

7. Whether or not an agent is a chair or PC member is readable by everyone. The authors of submitted papers are only readable by PC members.

8. The reviewers and sub-reviewers of a paper and whether a review is submitted is readable by all the PC members except the authors of that paper.

9. A PC member can read the review of paper p if (1) he is not the author of p (2) the review is already submitted (3) in the case that he is a reviewer or sub-reviewer of p, then he has submitted his own review.

The policy definition for the conference paper review system in PoliVer syntax is defined in figure 4.2. In the following, we present several queries for the policy in figure 4.2 in PoliVer query syntax.

Query 4.1.

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

The above run-statement declares that system contains 3 objects of type `Paper` and 4 objects of type `Agent`. The query asks if for some agent a which is not a chair, there exists a strategy in which a can promote himself to become the reviewer of a paper. PoliVer finds no strategy for this query.

Query 4.2.

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

This query looks for the strategy in which for two disjoint agents a and c and paper p, where c is a chair and a is not the author of p, agent c can allocate a as the reviewer of paper p. The output strategy of PoliVer shows that c can assign a as the PC member, and then appoint him as the reviewer of the paper p.

```
AccessControlSystem Conference
  Class Paper;
  Predicate
     author(paper:  Paper, agent:  Agent),
     pcmember(agent:  Agent), chair(agent:  Agent),
     reviewer(paper:  Paper, agent:  Agent),
     subreviewer(paper:  Paper, appointer:Agent, appointee:Agent),
     submittedreview(paper:  Paper, agent:  Agent),
     review(paper:  Paper, agent:  Agent);

  Action addPcmember(a:  Agent):  {+pcmember(a)} <- chair(user);
  Action delPcmember(a:  Agent):  {-pcmember(a)} <- pcmember(a) and a=user;
  Action addReviewer(p:  Paper, a:  Agent):
     {+reviewer(p, a)} <- chair(user) & pcmember(a) & ~author(p,a);
  Action delReviewer(p:  Paper, a:  Agent):
     {-reviewer(p, a), A b:  Agent.-subreviewer(p,user,b)} <-
     (pcmember(user) & user=a & reviewer(p,user));
  Action addSubreviewer(p:  Paper, a:  Agent, b:  Agent):
     {+subreviewer(p, a, b)} <- reviewer(p,a) & ~author(p,b) & user=a &
     ~(E d:  Agent [subreviewer(p,a,d) | subreviewer(p,d,b)]);
  Action delSubreviewer(p:  Paper, a:  Agent, b:  Agent):
     {-subreviewer(p, a, b)} <- subreviewer(p,a,b) &
     ~submittedreview(p,b) & user=b;
  Action submitreview(p:  Paper, a:  Agent):
     {+submittedreview(p, a)} <- (user=a) &
     ((E b:  Agent [subreviewer(p, b, user)]) |
     reviewer(p, user)) & ~submittedreview(p, user);
  Action addReview(p:  Paper, a:  Agent):
     {+review(p, a)} <- user=a & ((E b:  Agent [subreviewer(p, b, user)]) |
     reviewer(p,a)) & ~submittedreview(p, user);

  Read chair(a) <- true;
  Read pcmember(a) <- true;
  Read author(p, a) <- pcmember(user);
  Read reviewer(p, a) <- pcmember(user) & ~author(p, user);
  Read subreviewer(p, a, b) <- (pcmember(user) & ∼author(p,user)) |
     user=b | user=a;
  Read submittedreview(p, a) <- pcmember(user) & ∼author(p, user);
  Read review(p, a) <- 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))));
End
```

Figure 4.2: The policy definition for a conference paper review system. Note that PoliVer syntax for the rules is a simplified version of the original syntax presented in chapter 3

Query 4.3.

```
run for 3 Paper, 4 Agent
check {E dist a,b,c:Agent, p:Paper || chair(c)*!  & ∼author(p,a)*!  &
     submittedreview(p,b)*!  & ∼submittedreview(p,a)!  & pcmember(a)*!  &
     reviewer(p,a)!  & ∼subreviewer(p,b,a)*!  & ∼subreviewer(p,c,a)*!  &
     ∼subreviewer(p,a,a)*!  -> {a}:(<review(p,b)> THEN
     {a,c}:(submittedreview(p,a)))}
```

This nested query checks if for three disjoint agents a, b and c and paper p and in the case that c is a chair, a is the reviewer of p and b has submitted his review for p, it is possible for a to read the review that b has written for p and then, a and c collaborate in such a way that a submits his review for p.

Verification of the property by PoliVer results in a strategy that says: a first resigns as the reviewer of p and now, he is allowed to read the review that b has submitted. Then a allocates him as the reviewer of p again. In the next step, a submits his own review for paper p.

The variable assignment and output strategy of PoliVer is of the following form:

```
Assignment:  [a=1 b=2 c=3 p=1]
[1]:Agent 1 performs delReviewer(1,1);
     [1, 3]:Agent 3 performs addReviewer(1,1);
             Agent 1 performs submitreview(1,1);
             Goal;
```

As for the policy and query language, the syntax of the strategy in the implementation has some differences with the basic syntax in section 4.1.2. The output strategy consists of two sections as the query contains a nested goal. The agents in the square brackets are the coalitions that collaborate to achieve the goal. As the agent that performs the action is not included in the arguments, he is presented apart from the action in the strategy.

Query 4.4.

```
run for 1 Paper, 3 Agent
check {E dist a,c :Agent || chair(c)*!  & ∼chair(a)*!  & ∼pcmember(a)!)
     -> c:(pcmember(a) THEN a:(∼pcmember(a) THEN c:(pcmember(a) THEN
     a:(∼pcmember(a) THEN c:(pcmember(a))))))}
```

The main reason of presenting of the above query is to compare the efficiency with AcPeg in verifying highly nested queries. The query asks for the existence of any strategy that an agent can be assigned as a PC member and then resign, and this procedure continues multiple times. PoliVer shows that such strategy exists. The comparison of the time and memory usage between PoliVer and AcPeg will be presented in the next section.

A student information system (SIS) Figure 4.3 demonstrates the policy for a simple student information system written in PoliVar syntax. The rules of the system are as follows:

- A lecturer can appoint a student in a higher year as the demonstrator of a student in a lower year.
- The demonstrator of a student *s* can resign as being the demonstrator of *s*.
- The lecturers and demonstrators have the right to mark the students that are appointed to them.
- Whether an agent is a lecturer, demonstrator or student is readable by all the agents. Student marks are also readable by all the agents.

Query 4.5.

```
run for 8 Agents
check {E dist l, a1, a2:  Agent || lecturer(l)*!  & student(a1)*!  &
   student(a2)*!  & higher(a1,a2)*!  ->
   {l}:(demonstrator_of(a2,a1) & demonstrator_of(a1,a2))}
```

The query checks if there it is possible for a lecturer to assign two agents as the demonstrator of each other. Verifying this query in PoliVer returns no strategy.

```
AccessControlSystem StudentInformationSystem
  Class Agent;
  Predicate
     lecturer(agent:  Agent), student(agent:  Agent),
     demonstrator_of(demonstrator:  Agent, student:  Agent),
     higher(senior:  Agent, junior:  Agent),
     mark(student:  Agent);

  Action assignAsDem(d:Agent, s:Agent):
     {+demonstrator_of(d,s)} <- lecturer(user) & higher(d,s) & ∼higher(s,d);
  Action resignAsDem(d:Agent, s:Agent):
     {-demonstrator_of(d,s)} <- demonstrator_of(d,s) & user=d;
  Action MarkStudent(s:  Agent):
     {+mark(s)} <- lecturer(user) | demonstrator_of(user, s);

  Read higher(s,j) <- true;
  Read student(s) <- true;
  Read lecturer(l) <- true;
  Read mark(s) <- true;
End
```

Figure 4.3: The PoliVer policy script for student information system.

```
AccessControlSystem EmployeeInformationSystem
  Class Bonus;
  Predicate
     bonus(employee: Agent, bonus: Bonus), manager(employee: Agent),
     director(employee: Agent),
     advocate(appointer: Agent, appointee: Agent);

  Action addBonus(a: Agent, b: Bonus):
     {+bonus(a, b)} <- (manager(user) & ~manager(a) & ~director(a)) |
     director(user);
  Action delBonus(a: Agent, b: Bonus):
     {-bonus(a, b)} <- (manager(user) & ~manager(a) & ~director(a)) |
     director(user);
  Action addManager(a: Agent):
     {+manager(a)} <- director(user);
  Action delManager(a: Agent):
     {-manager(a)} <- user=a & manager(a) & ~director(a);
  Action addAdvocate(a1: Agent, a2: Agent):
     {+advocate(a1,a2)} <- user=a1;
  Action delAdvocate(a1: Agent, a2: Agent):
     {-advocate(a1,a2)} <- user=a2 and advocate(a1,a2);

  Read bonus(a, b) <- (user=a or director(user))
     | (manager(user) & ~manager(a) & ~director(a))
     | (advocate(a,user));
  Read manager(a) <- true;
  Read director(a) <- true;
  Read advocate(a1, a2) <- true;
End
```

Figure 4.4: The PoliVer policy script for employee information system.

An employee information system (EIS) The policy for an employee information system in PoliVer syntax is presented in figure 4.4. In the employee information system, a director can appoint an employee to become a manager. A manager can allocate bonuses of different options to other employees that are not managers or directors. A director can allocate bonuses to all the employees. The allocated bonuses is readable to the directors. The managers are able to access the allocated bonuses of the employees except the bonuses of other managers and directors. An employee can assign another employee as his advocate. An employee has access to the bonus information of his advocates.

Query 4.6.

```
run for 6 Bonus, 12 Agent
check {E dist a1,a2: Agent, b: Bonus || ∼director(a1)*! &
   ∼director(a2)*! & manager(a1)! & manager(a2)! & ∼bonus(a1,b)! ->
   {a1,a2}:(bonus(a1,b))}
```

The query checks if it is possible for two employees that are initially managers to collaborate in such a way that one of them can set the bonus for the other. PoliVer outputs a strategy that shows if one of the agents resigns as the manager, then the other one can set the bonus for him.

Query 4.7.

```
run for 6 Bonus, 12 Agent
check {dist a1,a2: Agent, b: Bonus || ∼director(a1)*! &
   ∼director(a2)*! & manager(a1)! & manager(a2)! & ∼bonus(a1,b)! ->
   a1,a2:(bonus(a1,b) & manager(a1))}
```

The query is similar to the query 4.6 except that it checks the case in which the employee with the bonus remains a manager. PoliVer finds no strategy for this query.

A password changing policy We recall the password changing policy in example 4.2 (figure 4.5) and add several actions that demonstrate joining end exiting some role, and make some dependencies between accessing some information and being the member of the roles.

Query 4.8.

```
run for 1 P, 2 Agent
check {A p: P, a: Agent || ∼roleA(p) & ∼roleB(p) -> {a}:(changePass(p))}
```

The output of the PoliVer contains two different strategies. In one of them, agent a needs to guess the value of password permission variable in order to find the appropriate path. In the other strategy, a can reach the goal without any risk. So, agent a is able choose the strategy that does not contain the risk of guessing the path.

```
Assignment:  [p=1 a=1]
[1]:Agent 1 performs roleBEnrol(1);Agent 1 performs roleAEnrol(1);
    (Guess:)  if (changePassPerm(1) is true){
              Agent 1 performs setChangePass(1);Goal;}
              else {Agent 1 performs setTrick(1);
              Agent 1 performs setChangePass(1);Goal;}


[1]:Agent 1 performs roleAEnrol(1);Agent 1 performs roleBEnrol(1);
    if (changePassPerm(1) is true){
       Agent 1 performs setChangePass(1);Goal;}
       else {Agent 1 performs setTrick(1);
       Agent 1 performs setChangePass(1);Goal;}
```

4.5 Experimental results

One of the outcomes of the implementation was the considerable reduction of binary decision diagram (BDD) variable size compared to RW. In RW, there are 7 knowledge states per proposition and therefore, an access control system with n propositions contains 7^n different states. Our simplification of knowledge-state variables results in 2^n states. The post-processing time for knowledge verification over found strategies is negligible compared to the whole process of strategy finding, while produces more expressive results.

We encoded authorization policies for a conference review system (CRS), employee information system (EIS) and student information system (SIS) into our policy language. We compared the performance in terms of verification time and memory usage for the queries: query 4.2 for CRS with 7 objects (3 papers and 4 agents) that looks for strategies which an agent can promote himself to become a reviewer of a paper, query 4.3 for CRS which is a nested query that asks if a reviewer can submit his review for a paper while he has read the review of someone else before, query 4.4 with 4 objects for CRS with five-level nested queries that checks if an agents can be assigned as a PC member by the chair and then resign his membership, query 4.5 for SIS with 10 objects that asks if a

```
AccessControlSystem PasswordPolicy
  Class P;
  Predicate
     roleA(p:  P), roleB(p:  P),
     changePassPerm(p:  P), trick(p:  P), changePass(p:  P);

  Action roleAEnrol(p:  P) : {+roleA(p)} <- true;
  Action roleAExit(p:  P) :{-roleA(p)} <- true;
  Action roleBEnrol(p:  P) :{+roleB(p)} <- true;
  Action roleBExit(p:  P) :{-roleB(p)} <- true;
  Action setTrick(p:  P) :
     {+trick(p)} <- ~changePassPerm(p) & roleA(p) & roleB(p);
  Action resetTrick(p:  P) :
     {-trick(p)} <- ~changePassPerm(p) & roleA(p) & roleB(p);
  Action setChangePass(p:  P) :
     {+changePass(p)} <- trick(p) | changePassPerm(p);

  Read roleA(p) <- true;
  Read roleB(p) <- true;
  Read changePassPerm(p) <- roleA(p) & ~roleB(p);
  Read trick(p) <- true;
  Read changePass(p) <- true;
End
```

Figure 4.5: The PoliVer policy script for changing password.

	RW(Algo-1)		PoliVer algorithm	
Query	Time	Memory	Time	Memory
Query 4.2	2.05	18.18	0.27	3.4
Query 4.3	0.46	9.01	0.162	6.68
Query 4.4	6.45	59.95	0.52	6.61
Query 4.5	20.44	222.02	0.488	7.30
Query 4.6	9.10	102.35	0.8	12.92

Figure 4.6: A comparison of query verification time (in second) and runtime memory usage (in MB) between RW and PoliVer.

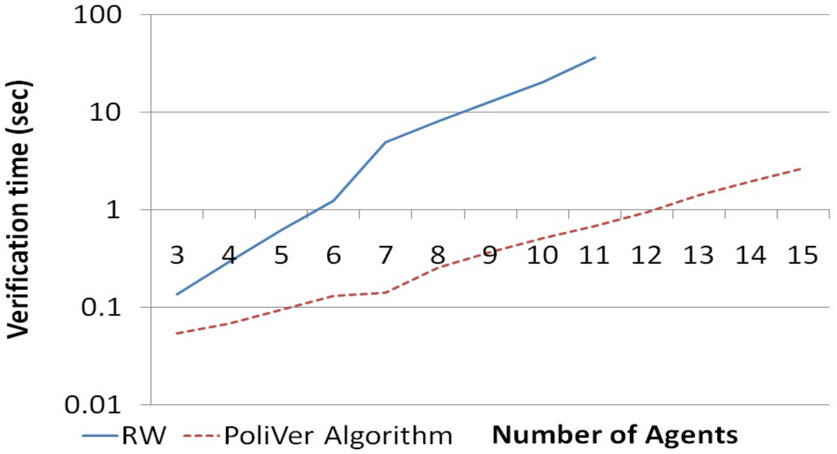

Figure 4.7: Verification time vs. number of agents for RW and PoliVer (query 4.5)

lecturer can assign two students as the demonstrator of each other, and query 5.4 with 18 objects for EIS which evaluates if two managers can collaborate to set a bonus for one of them .

Figure 4.6 shows a considerable reduction in time and memory usage by the proposed algorithm compared to Algo-1 in RW (Algo-1 has slightly better performance and similar memory usage compared to Algo-0). As a disadvantage for both systems, the verification time and state space grow exponentially when more objects are added. But this situation in PoliVer is much better than RW. Our experimental results demonstrate the correctness of our claim in practice by comparing the verification time of query 6.8 for different number of agents. Figure 4.7 sketches the verification time for both algorithms for different number of agents in logarithmic scale. The verification time in RW increases as 2.5^n where n is the number of agents added, while the time increases as 1.4^n in PoliVer. Note that this case study does not show the worst case behaviour when the number of agents increases[1].

[1] The tool and case studies are accessible at: `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/`

4.6 Summary

Our language and tool is optimised for analysing the access control policies of web-based collaborative systems such Facebook, LinkedIn and Easychair. These systems are likely to become more and more critical in the future, so analysing them is important. More specifically, in this work:

- We have developed a policy language and verification algorithm, which is also implemented as a tool. The algorithm produces evidence (in the form of a strategy) when the system satisfies a property.

- We remove the requirement to reason explicitly about knowledge, approximating it with the simpler requirement to reason about readability as it is sufficient in many cases. Compared to RW that has 7^n states, we have only 2^n states in our approach (where n is the number of propositions). Also, complicated properties can be evaluated over the policy by the query language provided.

- We detect the vulnerabilities in the policy that enable an attacker to discover the strategy to achieve the goal, when some required information is not accessible. We introduce the concept of effective propositions to detect such vulnerabilities.

- A set of propositions can be updated in one action. In the RW framework, each write action can update only one proposition at a time.

CHAPTER 5

REASONING ABOUT KNOWLEDGE IN ACCESS CONTROL SYSTEMS

In chapter 4, an access control policy verification tool (PoliVer) was introduced and implemented, which is able to verify agents' knowledge gained by reading system variables. This complies with one of the meanings of the knowledge in its ordinary language, which means that the agent *sees* the truth of a sentence when the question is present. The question that arises is: in a multi-agent system, does a principal gain knowledge only by directly reading system information? The answer is negative. Agent also knows a sentence when he consciously assents to it [76]. Reasoning is one of the ways that an agent gains knowledge about the information. Let us have a simple example:

Example 5.1. Assume a conference paper review system in which all the PC members have access to the number of the papers assigned to each reviewer. Further assume that each PC member can see the list of the papers assigned to the reviewers which does not contain the papers that he is the author of. Then if Alice is a PC member and the author of a submitted paper, she can find who the reviewer of her paper is by comparing the number of papers assigned to each reviewer with the number of the assigned papers of the reviewer that she has access to.

In this chapter, a method that is able to verify information leakage vulnerabilities through reasoning about agents' knowledge is provided. We use the concept of knowledge as in *epistemic modal logic.* Moreover, we propose an abstraction and refinement algorithm in order to reduce the verification time and memory usage when verifying safety properties.

5.1 Overview

Let's consider a conference paper review system like EasyChair or HotCRP. For the privacy and accountability requirement, the following properties need to hold in the policy:

- If Alice is the author of a submitted paper in the conference, there must be no way for her to find out who is the reviewer of her paper (privacy).

- If the profile information of a PC member is changed (by himself or someone else, like the chair of the conference), he will always know that his information has changed (accountability).

The above properties takes the *knowledge* of the agents into account. The knowledge can be gained by directly accessing the information, or by reasoning. Reasoning uses local accessible data to infer information. In section 5.6 we will show that there are some circumstances that accessible data like the number of the papers assigned to each reviewer in a conference paper review system may result in leaking unauthorized information. Such information leakage cannot be detected by the tools designed for verifying temporal properties or the tools that approximate knowledge by readability, like PoliVer and RW.

In this chapter, we explain how to use *interpreted systems* [45, 46] in order to model the access control system described by a policy. Using interpreted systems enables the verification of temporal and epistemic properties. Our modelling approach allows us to verify the knowledge gained by both reading and reasoning about information, which does not occur in other verification tools.

The price we pay for verifying epistemic properties is the large number of states and therefore, the occurrence of state explosion even in medium size systems. As another outcome of the research, we develop an automated method for abstraction and refinement of safety properties in CTLK (CTL with knowledge modality K) [73]. Our method applies counterexample guided refinement when the generated counterexample is tree-like [32]. In this work, we only discuss the counterexamples with finite length paths produced by verifying safety properties, but this approach can be extended to the paths of infinite length using unfolding mechanisms [30]. The proposed counterexample guided refinement method is generic, but we only demonstrate the applications in asynchronous access control policies.

This chapter is organized as follows: Interpreted systems are introduced in section 5.2, deriving an interpreted system from a policy is described in section 5.3, abstraction and refinement technique is given in sections 5.4, 5.4.2 and 5.5. Case studies and experimental results are included in section 5.6.

5.2 Background

5.2.1 Interpreted systems

Fagin et al. [46] introduced interpreted systems as the framework to model multi-agent systems in games scenarios. They introduced a detailed transition system which contains agents, local states and actions. Such a framework enables reasoning about both temporal and epistemic properties of the system. Lomuscio et al [75] have used a variant of interpreted systems to verify ATLK (alternating time temporal logic [5] with knowledge) properties over the interpreted systems. They have also developed a model-checker for interpreted systems called MCMAS [74, 72] which we will use as the model-checking engine in our implementation.

5.2.2 Definition of interpreted systems

The multi-agent system formalism known as *interpreted systems (IS)* [45, 46] contains a set $\Omega = \{e, 1, \ldots, n\}$ of agents including the *environment* e with the same specification as the other agents. Interpreted systems contain the following elements:

- **Local states:** Each agent in a multi-agent framework has its own local state. The set of local states for the agent i is denoted by L_i. The local state of an agent represents the information the agent has direct access to. The environment can be seen as the agent which is capable of capturing or holding the information that is inaccessible to the other agents. For example, the communication channel in a bit transmission protocol can be modelled as the environment. The set of *global states* is $S = L_e \times L_1 \times \cdots \times L_n$, representing the system at a specific time. The system evolves as a function over the time. We also use the notation of L_i as the function that accepts a set of global states and returns the corresponding set of local states for agent i. For each $s \in S$, $l_i(s)$ denotes the local state of agent i in s.

- **Actions:** State transitions are the result of performing actions by different agents. If $i \in \Omega$, then ACT_i is the set of actions accessible for the agent i. The set of *joint actions* is defined as $ACT = ACT_e \times ACT_1 \times \cdots \times ACT_n$. We also use ACT_i as the function that accepts a joint action and returns the action of agent i.

- **Protocols:** Protocols are defined as mappings from the set of local states to the set of local actions and define the actions each agent can perform according to its local state ($P_i : L_i \rightarrow 2^{ACT_i} \backslash \{\emptyset\}, i \in \Omega$). In general, action performance is non-deterministic.

We now provide the formal definition of interpreted systems based one the fundamental elements we have defined.

Definition 5.1 (Interpreted system)**.** Let Φ be a set of propositions and $\Omega = \{e, 1, \dots, n\}$ be a set of agents. An *interpreted system* I is a tuple:

$$I = \langle (L_i)_{i\in\Omega}, (P_i)_{i\in\Omega}, (ACT_i)_{i\in\Omega}, S_0, \tau, \gamma \rangle$$

where (1) L_i is the set of local states of agent i, and the set of global states is defined as $S = L_e \times L_1 \times \cdots \times L_n$ (2) ACT_i is the set of actions that agent i can perform, and $ACT = ACT_e \times ACT_1 \times \cdots \times ACT_n$ is defined as the set of joint actions (3) $S_0 \subseteq S$ is the set of initial states (4) $\gamma : S \times \Phi \to \{\top, \bot\}$ is called the *interpretation function* (5) $P_i : L_i \to 2^{ACT_i} \backslash \{\emptyset\}$ is the protocol for agent i (6) $\tau : ACT \times S \to S$ is called the *partial transition function* with the property that if $\tau(\alpha, s)$ is defined, then for all $i \in \Omega : \ ACT_i(\alpha) \in P_i(l_i(s))$. We also write $s_1 \overset{\alpha}{\to} s_2$ if $\tau(\alpha, s_1) = s_2$.

Definition 5.2 (Reachability)**.** A global state $s \in S$ is *reachable* in the interpreted system I if there exists $s_0 \in S_0$, $s_1, \dots, s_n \in S$ and $\alpha_1, \dots, \alpha_n \in ACT$ such that for all $1 \leq i \leq n : \ s_i = \tau(\alpha_i, s_{i-1})$ and $s = s_n$. In this chapter, we use G to denote the set of reachable states.

For an interpreted system I and each agent i we define an epistemic accessibility relation on the global states as follows:

Definition 5.3 (Epistemic accessibility relation)**.** Let I be an interpreted system and i be an agent. We define the *Epistemic accessibility relation for agent i*, written $\sim_i$, on the global states of I by

$$s \sim_i s' \quad \text{iff} \quad l_i(s) = l_i(s') \text{ and } s \text{ and } s' \text{ are reachable}$$

On the relation between interpreted systems and labelled transition systems: Interpreted systems are a class of labelled transitions systems specifically designed as a framework for formal verification of epistemic logic. To verify the temporal properties introduced in chapter 4, a simple labelled transition system is the appropriate model, which abstracts away the complexity of interpreted systems. We can prove that the labelled transition system in chapter 4 can be modelled by a special case of interpreted systems, which has the same semantic properties as the interpreted system which is derived from the same policy (see section 5.3). The proof is presented in appendix C.

5.2.3 CTLK logic

We use CTLK [73] as the specification language. CTL (Computational Tree Logic) is a branching-time temporal logic which has tree-like time model structure and allows quantification over paths, and CTLK adds the epistemic modality K to the CTL. CTLK is defined as follows:

Definition 5.4. Let Φ be a set of atomic propositions and Ω be a set of agents. If $p \in \Phi$ and $i \in \Omega$, then CTLK formulae are defined by:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid K_i\phi \mid EX\phi \mid EG\phi \mid E(\phi U \phi)$$

The symbol E is existential path quantifier which means "there exists at least one path"'. Temporal connectives X, G and U mean "neXt state", "all future states (Globally)" and "Until"'. EX, EG and EU provide the adequate set of CTLK connectives. For instance, safety properties defined by $AG(\phi)$ (all future states (Globally)) where A is the universal path quantifier, can be written as $\neg E(\top U \neg\phi)$, or the equivalence for liveness properties $AF(\phi)$ (always for some future state) is $\neg EG(\neg\phi)$. Epistemic connective K_i means "agent i knows that".

Example 5.2. Consider a conference paper review system. Assume that $\mathtt{a}_1$ is the author of the paper $\mathtt{p}_1$. Then the safety property that says if all the papers are assigned to the reviewers and $\mathtt{a}_2$ is the reviewer of $\mathtt{p}_1$, then $\mathtt{a}_1$ does not know the fact that $\mathtt{a}_2$ is the reviewer of his paper can be defined as: $AG(\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2) \rightarrow \neg K_{\mathtt{a}_1}\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2))$.

In an student information system, the property that states no two students can be assigned as the demonstrator of each other is specified by: $AG(\neg(\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3) \wedge \mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)))$.

Definition 5.5 (Satisfaction relation)**.** Let I be an interpreted system, $s \in G$ where G is the set of reachable states and $p \in \Phi$ where Φ is the set of atomic propositions. For any CTLK-formula ϕ, the notation $(I, s) \models \phi$ means ϕ holds at state s in interpreted system

I. The relation $\models$ is defined inductively as follows:

$$\begin{array}{lcl}
(I,s) \models p & \Leftrightarrow & \gamma(s,p) = \top \\
(I,s) \models \neg\phi & \Leftrightarrow & (I,s) \not\models \phi \\
(I,s) \models \phi_1 \vee \phi_2 & \Leftrightarrow & (I,s) \models \phi_1 \text{ or } (I,s) \models \phi_2 \\
(I,s) \models K_i\phi & \Leftrightarrow & (I,s') \models \phi \text{ for all } s' \in G \text{ such that } s \sim_i s' \\
(I,s) \models EX\phi & \Leftrightarrow & \text{for some } s' \text{ such that } s \overset{\alpha}{\to} s' : (I,s') \models \phi \\
(I,s) \models EG\phi & \Leftrightarrow & \text{there exists a path } s_1 \overset{\alpha}{\to} \ldots \text{ such that } s = s_0 \text{ and for all}
\end{array}$$

$$i \geq 0 : (I,s_i) \models \phi$$

$$\begin{array}{lcl}
(I,s) \models E(\phi_1 U \phi_2) & \Leftrightarrow & \text{there exists a path } s_1 \overset{\alpha}{\to} \ldots \text{ such that } s = s_1, \text{ there is}
\end{array}$$

$$\text{some } i \geq 1 \text{ such that } (I,s_i) \models \phi_2 \text{ and for all } j < i \text{ we have } (I,s_j) \models \phi_1$$

We use the notation $I \models \phi$ if for all $s_0 \in S_0$: $(I,s_0) \models \phi$.

5.3 Building an interpreted system from a policy

In access control systems, we deal with read and write access procedures. Write procedures, which update a set of variables, are contained in interpreted systems as actions. In interpreted systems, a principal knows a fact if it is included in his local state or he can deduce it by applying logical reasoning. In access control systems and in addition to the local information, agents may obtain permission to directly access some resources in the system. This permission may be granted by the system or other agents (delegation of authority). For instance, in a web application users always have access to their own profile, but they cannot access other users' profile unless the permission is granted by the owners. When a read permission to a resource is granted, the resource will become a part of agent's local state. When the permission is denied, it will be removed from agent's directly accessible information. This behaviour is similar to a system which uses dynamically changing local states to model permissions.

Interpreted systems formally contain local states which cannot change during execution of the system. In order to model temporary read permissions, we need to introduce some locally accessible information, which simulates the temporary read access. In this section, we explain how to introduce temporary read permissions when modelling access control systems. Moreover and as was discussed in chapter 3, we model access control systems in asynchronous manner using interpreted systems framework. An interpreted system is *asynchronous* if all joint actions contain at most one non-Λ agent action where Λ denotes no-operation.

Given a policy, we build an access control system based on interpreted systems framework by considering the requirements above. Incorporating temporary read permissions requires introducing some information into the local states. We say the proposition p is local to the agent i if its value only depends on the local state of i. In the other words, for all $s, s' \in S$ where $s \sim_i s'$ we have $\gamma(s,p) = \gamma(s',p)$.

Definition 5.6 (Local interpretation)**.** Let L_i be the set of local states of agent i in interpreted system I and Φ_i be the set of local propositions. We define the *local interpretation* for agent i as a function $\gamma_i : L_i \times \Phi_i \to \{\top, \bot\}$ such that $\gamma_i(l,p) = \gamma(s,p)$ where $l_i(s) = l$ for some global state s. We require the set of local propositions to be pairwise disjoint.

The following lemma provides the theoretical background of modelling knowledge by readability in an interpreted system.

Lemma 5.1. Let I be an interpreted system, G the set of reachable states, i an agent, Φ the set of propositions and $p \in \Phi$. Suppose that $p', p'' \in \Phi_i$. If for all $s \in G$:

$$\text{if } \gamma_i(l_i(s), p'') = \top \text{ then } (I,s) \models p \;\Leftrightarrow\; \gamma_i(l_i(s), p') = \top \tag{5.1}$$

Then we have:

$$\gamma_i(l_i(s), p'') = \top \quad \Rightarrow \quad (I,s) \models K_i p \vee K_i \neg p$$

Proof. We first prove that

$$\gamma_i(l_i(s), p'') = \top \text{ and } (I,s) \models p \quad \Rightarrow \quad (I,s) \models K_i p \tag{5.2}$$

Let us assume that $\gamma_i(l_i(s), p'') = \top$ and $(I,s) \models p$. By (5.1) we have $\gamma_i(l_i(s), p') = \top$. Consider any state $s_1 \in G$ such that $s_1 \sim_i s$. By the definition of $\sim_i$, we have $l_i(s_1) = l_i(s)$. Therefore, $\gamma_i(l_i(s_1), p') = \top$ and $\gamma_i(l_i(s_1), p'') = \top$ which implies $(I, s_1) \models p$. Hence, by the definition of K_i we are able to conclude that $(I,s) \models K_i p$. The proof for the second case:

$$\gamma_i(l_i(s), p'') = \top \text{ and } (I,s) \models \neg p \Rightarrow (I,s) \models K_i \neg p \tag{5.3}$$

is similar to the first proof. Therefore, by (5.2) and (5.3) we have $\gamma_i(l_i(s), p'') = \top \Rightarrow (I,s) \models K_i p \vee K_i \neg p$. □

To model knowledge by readability, we incorporate all the atomic propositions that appear in the policy into the environment. We call those propositions *policy propositions.* Now for each policy proposition p and for each agent, we introduce two local atomic propositions: p_{read} (p'' in Lemma 5.1) as the read permission of proposition p, and p_{loc} (p' in Lemma 5.1) as the local copy of p. We modify the transition function in order to satisfy the following property: for all reachable states, if p_{read} is `true` (agent has read access to p) in a state, then p_{loc} is assigned the same value as p. This property guarantees agent's knowledge of proposition p whenever his access to p is granted.

Building the interpreted system Given a policy $\mathcal{C}$ with Σ_{Ag} as the set of agents, we build up an interpreted system that models the access control system in the following way:

Let $\Phi_{\mathcal{C}}$ be the set of propositions that appear in $\mathcal{C}$ (policy propositions), and $\mathcal{A}_{\mathcal{C}}$ and $\mathcal{R}_{\mathcal{C}}$ the set of actions and read permissions in $\mathcal{C}$ respectively. For an interpreted system that corresponds to the policy $\mathcal{C}$, the knowledge gained by reading system information need to be incorporated into the local states of the agents.

Procedure 3 adopts Lemma 5.1 which describes a method to model temporary read permissions. The function incKnowledge in procedure 3 accepts $\mathcal{A}_{\mathcal{C}}$, $\mathcal{R}_{\mathcal{C}}$, $\Phi_{\mathcal{C}}$ and Σ_{Ag} as the input. For each agent i in Σ_{Ag}, Procedure 3 generates a set of local propositions Φ_i. The local state of agent i consists of all valuations of Φ_i. For each proposition $p \in \Phi_{\mathcal{C}}$, the set Φ_i contains two propositions p_{loc}, p_{read} where p_{loc} is the copy of p and gets updated whenever p_{read} as the access permission for p is `true` (refer to Lemma 5.1 for the details). The procedure modifies the actions and corresponding evolutions in $\mathcal{A}_{\mathcal{C}}$ into the set $\mathcal{A}^{u}_{\mathcal{C}}$ in order to update the propositions in Φ_i in the appropriate way. For each action and for each agent, if p appears in the effect (if-conditions in lines 12 and 18), then the action will replace with two freshly created actions: one sets p_{read} to `true` and p_{loc} to the same value as p if the read permission of p evaluates to `true` in the next state (lines 13 and 19). Otherwise (read permission of p evaluates to `false` in the next state), p_{read} will set to `false` and p_{loc} remains unchanged (lines 15 and 21). If p does not appear in the effect (line 24), p_{loc} and p_{read} will only get updated whenever the read permission of p is affected by the action.

Calculating the symbolic transition function: We provide the details for calculating the symbolic transition function we use for traversing over a path in our system. The symbolic transition function accepts a set of states as input and returns the result of performing an action over the states of that set.

As a convention, we use $s[p \mapsto m]$ where $s \in S$ to denote the state that is like s except that it maps the proposition p to the value m. Let $st \subseteq S$ be a set of states. When performing the action $\alpha : \varepsilon \leftarrow \ell$ in the states of st, the transition is only performed in the states that satisfy the permission ℓ. In the resulting states, the propositions that do not appear in ε remain the same as in the states that the transition begins. Therefore,

Procedure 3 Incorporating read permissions into evolution rules

```
function incKnowledge($\mathcal{A}_\mathcal{C}, \mathcal{R}_\mathcal{C}, \Phi_\mathcal{C}, \Sigma_{Ag}$)
    ▷ Input: $\mathcal{A}_\mathcal{C}$ is the set of actions, $\mathcal{R}_\mathcal{C}$ is the set of read permissions, $\Phi_\mathcal{C}$ the set of policy propositions and $\Sigma_{Ag}$ the set of agents
    ▷ Output: returns the updated set of actions and the set of local propositions
    $\mathcal{A}^u_\mathcal{C} := \mathcal{A}_\mathcal{C}$
    for all $i \in \Sigma_{Ag}$ do
        $\Phi_i := \emptyset$
        for all $p \in \Phi_\mathcal{C}$ do
            determine $r : p \leftarrow \ell_r \in \mathcal{R}_\mathcal{C}$ where $\mathbf{Ag}(r) = i$
            $\Phi_i := \Phi_i \cup \{p_{loc}, p_{read}\}$
            $\hat{\mathcal{A}}^u_\mathcal{C} := \emptyset$
            for all $\alpha : \varepsilon \leftarrow \ell \in \mathcal{A}^u_\mathcal{C}$ do
                if $+p \in \varepsilon$ then
                    construct $\alpha_1 : \varepsilon \cup \{+p_{loc}, +p_{read}\} \leftarrow$
                        $\ell \wedge (\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon])$ where $\mathbf{Ag}(\alpha_1) = \mathbf{Ag}(\alpha)$
                    construct $\alpha_2 : \varepsilon \cup \{-p_{read}\} \leftarrow$
                        $\ell \wedge \neg(\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon])$ where $\mathbf{Ag}(\alpha_2) = \mathbf{Ag}(\alpha)$
                    $\hat{\mathcal{A}}^u_\mathcal{C} := \hat{\mathcal{A}}^u_\mathcal{C} \cup \{\alpha_1, \alpha_2\}$
                else if $-p \in \varepsilon$ then
                    construct $\alpha_1 : \varepsilon \cup \{-p_{loc}, +p_{read}\} \leftarrow$
                        $\ell \wedge (\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon])$ where $\mathbf{Ag}(\alpha_1) = \mathbf{Ag}(\alpha)$
                    construct $\alpha_2 : \varepsilon \cup \{-p_{read}\} \leftarrow$
                        $\ell \wedge \neg(\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon])$ where $\mathbf{Ag}(\alpha_2) = \mathbf{Ag}(\alpha)$
                    $\hat{\mathcal{A}}^u_\mathcal{C} := \hat{\mathcal{A}}^u_\mathcal{C} \cup \{\alpha_1, \alpha_2\}$
                else
                    if for all $q \in \mathbf{fv}(\ell_r) : +q \notin \varepsilon$ and $-q \notin \varepsilon$ then
                        $\hat{\mathcal{A}}^u_\mathcal{C} := \hat{\mathcal{A}}^u_\mathcal{C} \cup \{\alpha\}$
                    else
                        construct $\alpha_1 : \varepsilon \cup \{+p_{loc}, +p_{read}\} \leftarrow \ell \wedge$
                            $(\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon]) \wedge p$ where $\mathbf{Ag}(\alpha_1) = \mathbf{Ag}(\alpha)$
                        construct $\alpha_2 : \varepsilon \cup \{-p_{loc}, +p_{read}\} \leftarrow \ell \wedge$
                            $(\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon]) \wedge \neg p$ where $\mathbf{Ag}(\alpha_2) = \mathbf{Ag}(\alpha)$
                        construct $\alpha_3 : \varepsilon \cup \{-p_{read}\} \leftarrow \ell \wedge$
                            $\neg(\ell_r[\top/v \mid +v \in \varepsilon][\bot/v' \mid -v' \in \varepsilon])$ where $\mathbf{Ag}(\alpha_3) = \mathbf{Ag}(\alpha)$
                        $\hat{\mathcal{A}}^u_\mathcal{C} := \hat{\mathcal{A}}^u_\mathcal{C} \cup \{\alpha_1, \alpha_2, \alpha_3\}$
                    end if
                end if
            end for
            $\mathcal{A}^u_\mathcal{C} := \hat{\mathcal{A}}^u_\mathcal{C}$
        end for
    end for
    return $\{\Phi_i \mid i \in \Sigma_{Ag}\}$, $\mathcal{A}^u_\mathcal{C}$
end function
```

we define:

$$\Theta_\alpha(st) = \Big\{ s[p \mapsto \top \mid +p \in \varepsilon][p \mapsto \bot \mid -p \in \varepsilon] \;\Big|\; s \in st, (I, s) \models \ell \Big\}$$

The symbolic (BDD-based) presentation of Θ_α is contained in appendix B.

Definition 5.7 (Derived interpreted system)**.** Let $\mathcal{C}$ be a policy with Σ_{Ag} as the set of agents, $\Phi_{\mathcal{C}}$ the set of policy propositions, and $\mathcal{A}^u_{\mathcal{C}}$ and Φ_i, $i \in \Sigma_{Ag}$ derived from procedure 3. Let $\Omega = \{e\} \cup \Sigma_{Ag}$ and $\Phi = \bigcup_{i \in \Omega} \Phi_i$ where $\Phi_e = \Phi_{\mathcal{C}}$. Then the interpreted system derived from policy $\mathcal{C}$ is:

$$I_{\mathcal{C}} = \langle (L_i)_{i\in\Omega}, (P_i)_{i\in\Omega}, (ACT_i)_{i\in\Omega}, S_0, \tau, \gamma \rangle$$

where

1. L_i is the set of local states of agent i, where each local state is a valuation of the propositions in Φ_i. The set of global states is defined as $S = L_e \times L_1 \times \cdots \times L_n$
2. $ACT_i = \{\alpha \in \mathcal{A}^u_{\mathcal{C}} \mid \mathbf{Ag}(\alpha) = i\} \cup \{\Lambda\}$ where Λ denotes *no operation*, and a joint action is a $|\Omega|$-tuple such that at most one of the elements is non-Λ (asynchronous interpreted system). For simplicity, *we denote a joint action with its non-Λ element*
3. $S_0 \subseteq S$ is the set of initial states
4. γ is the interpretation function over S and Φ. If $p \in \Phi_i$ then we have $\gamma(s, p) = \gamma_i(l_i(s), p)$
5. P_i is the protocol for agent i where for all $l \in L_i$: $P_i(l) = ACT_i$
6. τ is the transition function that is defined as follows: if α is a joint action (or simply, an action) and $s \in S$, then $\tau(\alpha, s) = s'$ if $\Theta_\alpha(\{s\}) = \{s'\}$.

The system that we derive from policy $\mathcal{C}$ is a special case of interpreted systems where the local states are the valuation of local propositions that are generated by the procedure incKnowledge.

5.4 Abstraction technique

In an interpreted system, the state space exponentially increases when extra propositions are added into the system. Considering a fragment of CTLK properties known as ACTLK as the specification language, we are able to apply abstraction and refinement techniques in order to verify the properties. ACTLK is defined as follows:

Definition 5.8. Let Φ be the set of atomic propositions and Ω set of agents. If $p \in \Phi$ and $i \in \Omega$, then ACTLK formulae are defined by:

$$\phi ::= p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid K_i\phi \mid AX\phi \mid A(\phi U \phi) \mid A(\phi R \phi)$$

where the symbol A is universal path quantifier which means "for all the paths".

In this section, we provide a brief introduction to the abstraction and refinement technique and the notations we use in this thesis. To provide a relation between the concrete model and the abstract one, we extend the *simulation relation* introduced in [31] to cover the epistemic relation between states. Using the abstraction technique that preserves simulation relation between the concrete model and the abstract one, we are able to verify ACTLK specification formulas over the model.

In this thesis and for abstraction and refinement, we focus on safety properties expressed in ACTLK. The advantages of safety properties are first, they are capable of expressing *policy invariants*, and second, the generated counterexample contains finite sequence of actions (or transitions). We can extend the abstraction refinement method to the full ACTLK by unfolding the loops in the counterexamples into finite transitions as described in [30], which is outside the scope of this chapter. Note that some properties like the first epistemic property in example 5.2 does not reside in the category of ACTLK and the abstraction and refinement procedure can not directly be applied to that formula. Later, we provide a method to verify such properties in an interactive manner.

5.4.1 Existential abstraction

The general framework of existential abstraction was first introduced by Clark et. al in [31]. Existential abstraction partitions the states of a model into clusters, or equivalence classes. The clusters form the states of the abstract model. The transitions between the clusters in the abstract model give rise to an over-approximation of the original (or concrete) model that *simulates* the original one. So, when a specification in ACTL (or in the context of this paper, ACTLK) logic is true in the over-approximated model, it will be true in the concrete one. Otherwise, a counterexample will be generated which needs to be verified over the concrete model.

Notation 5.1. For simplicity, we use the same notation ($\sim_i$) for the epistemic accessibility relation in both the concrete and abstract interpreted systems.

Definition 5.9 (Simulation)**.** Let I and $\widetilde{I}$ be two interpreted systems, Ω be the set of agents in both systems, and Φ and $\widetilde{\Phi}$ the corresponding set of propositions where $\widetilde{\Phi} \subseteq \Phi$. The relation $H \subseteq S \times \widetilde{S}$ is *simulation relation* between I and $\widetilde{I}$ if and only if:

1. For all $s_0 \in S_0$, there exists $\widetilde{s}_0 \in \widetilde{S}_0$ such that $(s_0, \widetilde{s}_0) \in H$.

and for all $(s, \widetilde{s}) \in H$:

2. For all $p \in \widetilde{\Phi}$: $\gamma(s, p) = \widetilde{\gamma}(\widetilde{s}, p)$

3. For each state $s' \in S$ such that $\tau(s, \alpha) = s'$ for some $\alpha \in ACT$, there exists $\widetilde{s}' \in \widetilde{S}$ and $\widetilde{\alpha} \in \widetilde{ACT}$ such that $\widetilde{\tau}(\widetilde{s}, \widetilde{\alpha}) = \widetilde{s}'$ and $(s', \widetilde{s}') \in H$.

4. For each state $s' \in S$ such that $s \sim_i s'$, there exists $\widetilde{s}' \in \widetilde{S}$ such that $\widetilde{s} \sim_i \widetilde{s}'$ and $(s', \widetilde{s}') \in H$.

The above definition for simulation relation over the interpreted systems is similar to the one for Kripke model [30], except that the relation for the epistemic relation is introduced. If such simulation relation exists, we say that $\widetilde{I}$ *simulates* I (denoted by $I \preceq \widetilde{I}$).

If H is a function, that is, for each $s \in S$ there is a unique $\widetilde{s} \in \widetilde{S}$ such that $(s, \widetilde{s}) \in H$, we write $h(s) = \widetilde{s}$ instead of $(s, \widetilde{s}) \in H$.

Lemma 5.2. Let $I \preceq \widetilde{I}$, $s_1 \in S$, $\widetilde{s}_1 \in \widetilde{S}$ and $(s_1, \widetilde{s}_1) \in H$ where H is the simulation relation between I and $\widetilde{I}$. Then for each path $s_1 \xrightarrow{\alpha_2} \ldots$ in I, there exists a path $\widetilde{s}_1 \xrightarrow{\widetilde{\alpha}_2} \ldots$ in $\widetilde{I}$ such that for all $i \geq 1$, $(s_i, \widetilde{s}_i) \in H$ holds.

Proof. The proof is trivial by item 3 in definition 5.9 and induction over the state transitions. □

Proposition 5.1. For every ACTLK formula φ over propositions $\widetilde{\Phi}$, if $I \preceq \widetilde{I}$ and $\widetilde{I} \models \varphi$, then $I \models \varphi$.

Proof. To prove the proposition, we first prove if $I \preceq \widetilde{I}$ and H is the simulation relation, then for all $\widetilde{s} \in \widetilde{S}$ and $s \in S$ where $(s, \widetilde{s}) \in H$, $(\widetilde{I}, \widetilde{s}) \models \varphi$ implies $(I, s) \models \varphi$. We assume φ is in NNF. The proof proceeds by induction over the structure of φ. Let $s \in S$, $\widetilde{s} \in \widetilde{S}$ and $(s, \widetilde{s}) \in H$.

- If $(\widetilde{I}, \widetilde{s}) \models p$ where p an atomic formula, then $\gamma(\widetilde{s}, p) = \top$. By item 2 in definition 5.9 we have $\gamma(s, p) = \top$ which implies $(I, s) \models p$. The case is similar for $\varphi = \neg p$.

- If $(\widetilde{I}, \widetilde{s}) \models \varphi_1 \wedge \varphi_2$, then $(\widetilde{I}, \widetilde{s}) \models \varphi_1$ and $(\widetilde{I}, \widetilde{s}) \models \varphi_2$. By induction hypothesis we have $(I, s) \models \varphi_1$ and $(I, s) \models \varphi_2$. Therefore, $(I, s) \models \varphi_1 \wedge \varphi_2$. The case is similar for $\varphi = \varphi_1 \vee \varphi_2$.

- Assume $(\widetilde{I}, \widetilde{s}) \models AX\varphi_1$. If $s \xrightarrow{\alpha} s'$ is a path in I, then by Lemma 5.2 there exists a path $\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}'$ in $\widetilde{I}$ where $(s', \widetilde{s}') \in H$. By the assumption we have $(\widetilde{I}, \widetilde{s}') \models \varphi_1$. Then the induction hypothesis implies $(I, s') \models \varphi_1$. Thus we can conclude that $(I, s) \models AX\varphi_1$.

- Assume $(\widetilde{I}, \widetilde{s}) \models A(\varphi_1 U \varphi_2)$. Let $s_1 \xrightarrow{\alpha_2} \ldots$ be a path in I where $s_1 = s$ and $\widetilde{s}_1 \xrightarrow{\widetilde{\alpha}_2} \ldots$ the corresponding path in $\widetilde{I}$ where $\widetilde{s}_1 = \widetilde{s}$. By the assumption, there exists some $i \geq 1$ where $(\widetilde{I}, \widetilde{s}_i) \models \varphi_2$ and $(\widetilde{I}, \widetilde{s}_i) \models \varphi_1$ for all $j < i$. By induction hypothesis and Lemma 5.2, $(I, s) \models \varphi_1 U \varphi_2$. As this property holds for all the path starting at s, we can conclude $(I, s) \models A(\varphi_1 U \varphi_2)$.

- Assume $(\widetilde{I}, \widetilde{s}) \models A(\varphi_1 R \varphi_2)$. The proof is similar to the case for $(\widetilde{I}, \widetilde{s}) \models A(\varphi_1 U \varphi_2)$.

- Assume $(\widetilde{I}, \widetilde{s}) \models K_i \varphi$. We pick a state $s' \in S$ where $s' \sim_i s$. By item 4 in definition 5.9, there exists $\widetilde{s}' \in \widetilde{S}$ where $\widetilde{s}' \sim_i \widetilde{s}$ and $(s', \widetilde{s}') \in H$. By the assumption, $(\widetilde{I}, \widetilde{s}') \models \varphi$. Induction hypothesis implies that $(I, s') \models \varphi$. As this property holds for all the states with accessibility relation $\sim_i$ to s, we have $(I, s') \models K_i \varphi$.

Now, if $\widetilde{I} \models \varphi$ or in the other words, for all $\widetilde{s}_0 \in \widetilde{S}_0$: $(\widetilde{I}, \widetilde{s}) \models \varphi$, then by item 1 in definition 5.9 and the above proof we have for all $s_0 \in S_0$: $(I, s) \models \varphi$ or equivalently $I \models \varphi$.

□

5.4.2 Variable hiding abstraction

Variable hiding is a popular technique in the category of existential abstraction. In our methodology, we consider factorizing the concrete state space into equivalence classes that act as abstract states by abstracting away a set of system propositions. In our approach, the states in each equivalence class are only different in the valuation of the hidden propositions. Also the transitions between the states of the abstract model are defined in such a way that the abstract model simulates the concrete one. Our refinement procedure will be splitting the abstract states by putting back some of the atomic proportions that were hidden in the abstract model. We refine the model by analysing the counterexample generated when verifying safety properties described in ACTLK logic. The model checker will output a counterexample if the property does not hold.

Definition 5.10. (Local state relation) Let $I_{\mathcal{C}}$ be an interpreted system derived from policy $\mathcal{C}$, L_i and Φ_i be the set of local states and local propositions for the agent i, and $\widetilde{\Phi}_i \subseteq \Phi_i$. The local relation $\Re_i$ is defined as:

$$\text{for all } l_1, l_2 \in L_i : \qquad l_1 \Re_i l_2 \quad \text{iff} \quad \text{for all } p \in \widetilde{\Phi}_i : \gamma_i(l_1, p) = \gamma_i(l_2, p)$$

where γ_i is the local interpretation for the agent i. The function $h_i : L_i \to L_i/\Re_i$ is the surjection which maps elements of L_i into equivalence classes of $\Re_i$.

Definition 5.11 (Action classification)**.** Let $\alpha : \varepsilon \leftarrow \ell \in ACT$ and $\widetilde{\Phi} \subseteq \Phi$. We define $\alpha' : \varepsilon' \leftarrow \ell' \in [\alpha]$ iff $\{\pm p \in \varepsilon' \mid p \in \widetilde{\Phi}\} = \{\pm p \in \varepsilon \mid p \in \widetilde{\Phi}\}$, $\exists(\Phi\backslash\widetilde{\Phi}).\ell' \equiv \exists(\Phi\backslash\widetilde{\Phi}).\ell$ and $\mathbf{Ag}(\alpha') = \mathbf{Ag}(\alpha)$.

In the above definition, the infix notation $\equiv$ denotes the semantically equivalence relation. Formally $\exists x.f$ for a Boolean function f is defined as $f[0/x] \vee f[1/x]$ which means f could be made to true by putting x to 0 or to 1. If $X = \{x_1, \dots, x_n\}$, then $\exists X.f = \exists x_1 \dots \exists x_n.f$.

Definition 5.12 (Abstract interpreted system)**.** Given a policy $\mathcal{C}$, let Ω, Φ and $\mathcal{A}^u_{\mathcal{C}}$ be deduced as described in section 5.3 and $I_{\mathcal{C}}$ be the derived interpreted system. Let $\widetilde{\Phi} \subseteq \Phi$ and $\widetilde{\Omega} = \Omega$. We define Interpreted system $\widetilde{I_{\mathcal{C}}}$ as:

$$\widetilde{I_{\mathcal{C}}} = \langle (\widetilde{L_i})_{i\in\widetilde{\Omega}}, (\widetilde{P_i})_{i\in\widetilde{\Omega}}, (\widetilde{ACT}_i)_{i\in\widetilde{\Omega}}, \widetilde{S}_0, \widetilde{\tau}, \widetilde{\gamma} \rangle$$

where

1. $\widetilde{L_i} = L_i/\Re_i$ where $\Re_i$ is defined in definition 5.10 over L_i, and $\widetilde{S} = \widetilde{L}_e \times \widetilde{L}_1 \times \cdots \times \widetilde{L}_n$
2. $\widetilde{ACT}_i = \{[\alpha] \mid \alpha \in \mathcal{A}^u_{\mathcal{C}}$ and $\mathbf{Ag}(\alpha) = i\}$ and a joint action is a $|\widetilde{\Omega}|$-tuple such that at most one of the elements is non-Λ - i.e. the system is asynchronous. As before, each joint action is shown by its non-Λ element. If $\widetilde{\alpha} = [\alpha]$, then the evolution rule for $\widetilde{\alpha}$ is $\widetilde{\varepsilon} \leftarrow \widetilde{\ell}$ where $\widetilde{\varepsilon} = \{\pm p \in \varepsilon \mid p \in \widetilde{\Phi}\}$ and $\widetilde{\ell} = \exists(\Phi\backslash\widetilde{\Phi}).\ell$
3. $\widetilde{S}_0 = \{(h_i(l_i(s)))_{i\in\widetilde{\Omega}} \mid s \in S_0\}$ where h_i as in definition 5.10 maps the elements of L_i to $\widetilde{L}_i$
4. For all $\widetilde{l} \in \widetilde{L}_i$ and for all $p \in \widetilde{\Phi}_i$ we have $\widetilde{\gamma}_i(\widetilde{l}, p) = \gamma_i(l, p)$ where $\widetilde{l} = h_i(l)$
5. $\widetilde{P_i}$ is the protocol for agent i where for all $\widetilde{l} \in \widetilde{L}_i$: $\widetilde{P_i}(\widetilde{l}) = \widetilde{ACT}_i$
6. $\widetilde{\tau}$ is the transition function defined as follows: If $\widetilde{\alpha}$ is a joint action, $\widetilde{s} \in \widetilde{S}$ and $\widetilde{\Theta}_{\widetilde{\alpha}}$ is the symbolic transition function for interpreted system $\widetilde{I_{\mathcal{C}}}$ and action $\widetilde{\alpha}$, then $\widetilde{\tau}(\widetilde{\alpha}, \widetilde{s}) = \widetilde{s}'$ if $\widetilde{\Theta}_{\widetilde{\alpha}}(\{\widetilde{s}\}) = \{\widetilde{s}'\}$

Proposition 5.2. If $I_{\mathcal{C}}$ is the interpreted system derived from policy $\mathcal{C}$ and $\widetilde{I_{\mathcal{C}}}$ is defined as in definition 5.12, then $I_{\mathcal{C}} \preceq \widetilde{I_{\mathcal{C}}}$.

Proof. Let $h : S \rightarrow \widetilde{S}$ be a function where $h(s) = (h_i(l_i(s)))_{i\in\widetilde{\Omega}}$ and h_i is defined as in definition 5.10. We show that $\widetilde{I_{\mathcal{C}}}$ simulates $I_{\mathcal{C}}$ under h. Item 1 in definition 5.9 trivially holds by property (3). Item 2 holds by property (4) and the fact that if $p \in \widetilde{\Phi}$, then there is an agent i where $p \in \widetilde{\Phi}_i$ and we have $\widetilde{\gamma}(\widetilde{s}, p) = \widetilde{\gamma}_i(\widetilde{l_i}(\widetilde{s}), p)$.

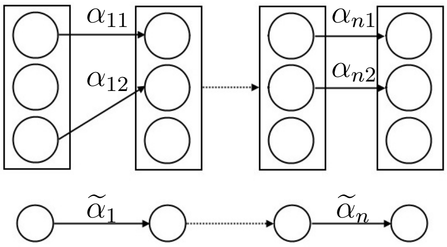

Figure 5.1: The counterexample provided by the abstract model may not be valid on the concrete one. The labels represent the actions that result in the transitions.

Now assume that $h(s) = \widetilde{s}$ and $\tau(\alpha, s) = s'$, which is equivalent to $\Theta_\alpha(\{s\}) = \{s'\}$. If $\alpha : \varepsilon \leftarrow \ell$ can be performed in s, then we have $(I, s) \models \ell$. It is trivial to show that $(I, s) \models \exists(\Phi\backslash\widetilde{\Phi}).\ell$ using structural induction. Since the formula $\exists(\Phi\backslash\widetilde{\Phi}).\ell$ only contains the propositions in $\widetilde{\Phi}$, then by item 2 in definition 5.9 we have $(\widetilde{I}, \widetilde{s}) \models \exists(\Phi\backslash\widetilde{\Phi}).\ell$. Let $\widetilde{\alpha} = [\alpha]$. By definition 5.11, $\widetilde{\alpha}$ can be performed in $\widetilde{s}$. From $\widetilde{\varepsilon} \subseteq \varepsilon$ we infer that the performance of $\widetilde{\alpha}$ on $\widetilde{s}$ results in a state $\widetilde{s'}$ where all the propositions in $\widetilde{\Phi}$ have the same value in $\widetilde{s'}$ as in s'. Hence, $h(s') = \widetilde{s'}$ as required for item 3 in definition 5.9.

Let us assume that $h(s) = \widetilde{s}$ and $s \sim_i s'$. Therefore $l_i(s) = l_i(s')$ which means that for all $p \in \Phi_i :\ \gamma(s, p) = \gamma(s', p)$. Since $\widetilde{\Phi} \subseteq \Phi$, then $\widetilde{\Phi}_i \subseteq \Phi_i$. By item 2 in definition 5.9, for all $p \in \widetilde{\Phi}_i :\ \gamma(s, p) = \widetilde{\gamma}(\widetilde{s}, p)$. Let us assume that $h(s') = \widetilde{s'}$. Then for all $p \in \widetilde{\Phi}_i :\ \gamma(s', p) = \widetilde{\gamma}(\widetilde{s'}, p)$. Hence we have for all $p \in \widetilde{\Phi}_i :\ \widetilde{\gamma}(\widetilde{s}, p) = \widetilde{\gamma}(\widetilde{s'}, p)$. Therefore $\widetilde{s} \sim_i \widetilde{s'}$ as required for item 4. □

Definition 5.13. We define $h_A : ACT \to \widetilde{ACT}$ as the surjection that maps the actions in the concrete model to the actions in the abstract one.

Given a policy, by using Proposition 5.2 we can build up an abstract access control system by hiding a set of propositions and abstracting the evolution rules. Now by proposition 5.1, it is possible to verify ACTLK properties over the abstract model, and refine the abstraction, if the property does not hold and the counterexample is found to be spurious.

5.5 Automated Refinement

Our counterexample based abstraction refinement method consists of three steps:

- *Generating the initial abstraction*: It is done by examining transition blocks corresponding to the variables and constructing clusters of variables which interfere with each other via transition conditions. In our approach, we build the simplest possible initial abstract model by only retaining only the propositions appear in specification φ that we aim to verify.

- *Model-checking the abstract structure*: Model-checking will be performed on the abstract model for a specification φ. If the abstract model satisfies φ, then it can be concluded that the concrete model also satisfies φ. If the abstract model checking generates a counterexample, it should be checked if the counterexample is an actual counterexample for the concrete model. If it is a spurious counterexample in the concrete model as in figure 5.1, the abstract system should be refined by proceeding to the next step.

- *Refining the abstraction*: The counterexample guided framework refines the abstract model by partitioning the states in abstract model in such a way that the refined model does not admit the same counterexample. For the refinement, we turn some of the invisible variables into visible. After refinement of the abstract model, step 2 will be proceeded.

The process of abstraction and refinement will eventually terminate, as in the worst case, the refined model becomes the same as the concrete one, which is a finite state model. Therefore in the worst case, the verification will turn into the verification of the concretised model.

5.5.1 Generating the initial abstraction

For automatic abstraction refinement, we build the initial model as simple as possible. For an ACTLK formula φ, we keep all the atomic propositions that appear in φ visible in the abstract model and hide the rest. The abstract model is built up by definition 5.12.

5.5.2 Validation of counterexamples

The structure of a counterexample created by the verification of an ACTLK formula is different from the counterexample generated in the absence of knowledge modality. In an ACTLK counterexample, we have epistemic relations as well as temporal ones. Analysis of such counterexamples is more complicated than the counterexamples for temporal properties.

A counterexample for a safety property in ACTLK is a loop-free tree-like graph with states as vertices, and temporal and epistemic transitions as edges. Every counterexample has an initial state as the root. A temporal transition in the graph is labelled with its

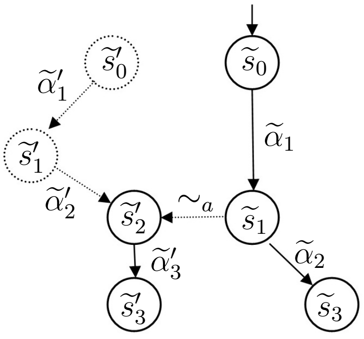

Figure 5.2: A tree-like counterexample generated by the verification of an ACTLK safety property over the abstract model. In the diagram, $\widetilde{s}_0, \widetilde{s}'_0 \in S_0$ and $\widetilde{s}_1 \sim_a \widetilde{s}'_2$. As reachability is a requirement for $\widetilde{s}_1 \sim_a \widetilde{s}'_2$ and $\widetilde{s}_1$ is already reachable, the temporal path $\widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \widetilde{s}'_1 \xrightarrow{\widetilde{\alpha}'_2} \widetilde{s}'_2$ provides the witness for the reachability of $\widetilde{s}'_2$. Considering this witness is required in counterexample checking.

corresponding action and epistemic transition is labelled with the corresponding epistemic relation. We define a *temporal path* as a path that contains only temporal transitions. An *epistemic path* contains at least one epistemic transition. Every state in the counterexample is *reachable from an initial state* in the model, which may differ from the root. For any state s, we write also s for the empty path which starts and finishes in s.

Counterexample formalism: A tree is a finite set of temporal and epistemic paths with an initial state as the root. Each path begins from the root and finishes at a leaf. For an epistemic transition over a path, we use the same notation as the epistemic relation while we consider the transition to be from left to the right. For instance, the tree in the figure 5.2 is formally presented by:

$$\{\widetilde{s}_0 \xrightarrow{\widetilde{\alpha}_1} \widetilde{s}_1 \xrightarrow{\widetilde{\alpha}_2} \widetilde{s}_3,\ \widetilde{s}_0 \xrightarrow{\widetilde{\alpha}_1} \widetilde{s}_1 \sim_a \widetilde{s}'_2 \xrightarrow{\widetilde{\alpha}'_3} \widetilde{s}'_3\}$$

To verify a tree-like counterexample, we traverse the tree in a *depth-first* manner. An abstract counterexample is valid in the concrete model if a real counterexample in the concrete model corresponds to it.

We use the notation $s \to s'$ when the type of the transition from s to s' is not known.

Definition 5.14 (Vertices, root)**.** Let $\widetilde{ce}$ be a counterexample. Then $\mathbf{Vert}(\widetilde{ce})$ denotes the set of all the states that appear in $\widetilde{ce}$. $\mathbf{Root}(\widetilde{ce})$ denotes the root of $\widetilde{ce}$. For a path $\widetilde{\pi}$, $\mathbf{Root}(\widetilde{\pi})$ denotes the state that $\widetilde{\pi}$ starts with.

Definition 5.15 (Corresponding paths)**.** Let $\widetilde{I}$ be an abstract model of the interpreted system I, h be the abstraction function, and h_A be the function that maps the actions in I to the ones in $\widetilde{I}$. The concrete path $\pi = s_1 \to \cdots \to s_n$ in the concrete model corresponds to the path $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ in the abstract model, if

$$\textsc{TemporalCheck}\ \frac{h_A^{-1}(\widetilde{\alpha}) = \{\alpha_1, \dots, \alpha_n\}}{(\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}' \,||\, \pi, st) \Rightarrow_t (\pi, \bigcup_{i=1}^{n} \Theta_{\alpha_i}(st) \cap h^{-1}(\widetilde{s}'))}$$

$$\textsc{EpistemicCheck}\ \frac{\begin{array}{c}\pi' = \widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \dots \xrightarrow{\widetilde{\alpha}'_m} \widetilde{s}' \text{ is a temporal path to } \widetilde{s}' \text{ where } \widetilde{s}'_0 \in \widetilde{S}_0 \\ (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^* (\widetilde{s}', st') \qquad \hat{st} = \{s \in st' \mid l_a(s) \in L_a(st)\}\end{array}}{(\widetilde{s} \sim_a \widetilde{s}' \,||\, \pi, st) \Rightarrow_e (\pi, \hat{st})}$$

Figure 5.3: Temporal and epistemic transition rules. In EpistemicCheck rule, π' is the witness for the reachability of $\widetilde{s}'$ in the abstract model, and st' is the concrete states that are reachable through the concrete paths corresponding to π'. In the case that the model-checker returns all the abstract paths to $\widetilde{s}'$, let us say $\widetilde{\Pi}'$, then st' will be calculated as $st' = \bigcup\{st \mid \pi' = \widetilde{s}'_0 \to \dots \to \widetilde{s}' \in \widetilde{\Pi}', \widetilde{s}'_0 \in \widetilde{S}_0 \text{ and } (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^* (\widetilde{s}', st)\}$.

- For all $1 \leq i \leq n : \widetilde{s}_i = h(s_i)$
- If $\widetilde{s}_i \xrightarrow{\widetilde{\alpha}_{i+1}} \widetilde{s}_{i+1}$ is a temporal transition, we have $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ where $h_A(\alpha_{i+1}) = \widetilde{\alpha}_{i+1}$.
- If $\widetilde{s}_i \sim_a \widetilde{s}_{i+1}$ is an epistemic transition, we have $s_i \sim_a s_{i+1}$ and s_{i+1} is reachable in the concrete model.

Definition 5.16 (Concrete counterexample)**.** Let $\widetilde{ce}$ be a tree-like counterexample in the abstract model where $\mathbf{Root}(\widetilde{ce}) \in \widetilde{S}_0$. A concrete counterexample ce corresponds to $\widetilde{ce}$ if $\mathbf{Root}(ce) \in S_0$ and there exists a one-to-one correspondence between the states and the paths of the counterexamples ce and $\widetilde{ce}$ according to the definition 5.15.

To *verify a path* in the counterexample, we define two transition rules TemporalCheck and EpistemicCheck denoted by $\Rightarrow_t$ and $\Rightarrow_e$ as in figure 5.3. For a path with the transition $\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}'$ as the head and for the concrete states st, the rule $\Rightarrow_t$ finds all the successors of the states in st which reside in $h^{-1}(\widetilde{s}')$. If the head of the path is the epistemic transition $\widetilde{s} \sim_a \widetilde{s}'$, then the rule $\Rightarrow_e$ extracts all the *reachable states* in $h^{-1}(\widetilde{s}')$ corresponding to π' as the witness of reachability of $\widetilde{s}'$, which has common local states with some states in $st \subseteq h^{-1}(\widetilde{s})$. Both the temporal and epistemic rules are deterministic.

Definition 5.17. We write $\Rightarrow_t^*$ to denote a sequence of temporal transitions $\Rightarrow_t$. We use $\Rightarrow^*$ to denote a sequence of the transitions $\Rightarrow_t$ or $\Rightarrow_e$.

Proposition 5.3 (Soundness of $\Rightarrow_t^*$)**.** Let $\widetilde{\pi}$ be a temporal path in the abstract model which starts at $\widetilde{s}_1$ and ends in $\widetilde{s}_n$. If $st_1 \subseteq h^{-1}(\widetilde{s}_1)$ and $(\widetilde{\pi}, st_1) \Rightarrow_t^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$, then there exists a concrete path that starts from a state in st_1 and ends in a state in st_n.

Proof. We use induction over the length of the path.

Base case: $\widetilde{\pi} = \widetilde{s}_1$. Then there is no transition from $(\widetilde{s}_1, st_1)$ and therefore, the concrete path is a state in st_1.

Inductive case: Assume by inductive hypothesis that for all $\widetilde{\pi} = \widetilde{s}_i \xrightarrow{\widetilde{\alpha}_{i+1}} \dots \xrightarrow{\widetilde{\alpha}_{i+k}} \widetilde{s}_{i+k}$ of length k, if $(\widetilde{\pi}, st_i) \Rightarrow_t^* (\widetilde{s}_{i+k}, st_{i+k})$ for some $st_i, st_{i+k} \subseteq S$, then there exists a concrete path which begins at a state in st_i and ends in a state in st_{i+k}. Consider that $\widetilde{\pi}' = \widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}$ is a path of the length $k+1$ where $(\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_t (\widetilde{\pi}, st_i) \Rightarrow_t^* (\widetilde{s}_{i+k}, st_{i+k})$. By induction hypothesis, there exists a concrete path that begins at some state $s_i \in st_i$ and ends in $s_{i+k} \in st_{i+k}$. By the definition of $\Rightarrow_t$, every state in st_i is the successor of some states in st_{i-1}. Therefore, there exists $s_{i-1} \in st_{i-1}$ and $\alpha_i \in h_A^{-1}(\widetilde{\alpha}_i)$ such that $\{s_i\} = \Theta_{\alpha_i}(\{s_{i-1}\})$. So we select the corresponding transition in the concrete model to be $s_{i-1} \xrightarrow{\alpha_i} s_i$ which allows s_{i-1} to reach s_{i+k} by the existence of a concrete path from s_i to s_{i+k}. □

By proposition 5.3 and definition 5.16, if $\widetilde{\pi} = \widetilde{s}_0 \xrightarrow{\widetilde{\alpha}_1} \dots \xrightarrow{\widetilde{\alpha}_n} \widetilde{s}_n$ is a path in the counterexample where $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow_t^* (\widetilde{s}_n, st_n)$, then there exists a corresponding concrete path beginning at an initial state $s_0 \in S_0 \cap h^{-1}(\widetilde{s}_0)$ which ends at some state $s_n \in st_n$.

Proposition 5.4 (Soundness of $\Rightarrow^*$)**.** Let $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ be a path in the abstract model. If $st_1 \subseteq h^{-1}(\widetilde{s}_1)$ and $(\widetilde{\pi}, st_1) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$, then there exists a concrete path that starts from a state in st_1 and ends in a state in st_n.

Proof. For the general form of a path that contains both temporal and epistemic transitions, we use the similar approach as in proposition 5.3.

Base case: $\widetilde{\pi} = \widetilde{s}_1$. Then there is no transition from $(\widetilde{s}_1, st_1)$ and therefore, the concrete path is a state in st_1.

Inductive case: Assume by inductive hypothesis that for all $\widetilde{\pi} = \widetilde{s}_i \to \cdots \to \widetilde{s}_{i+k}$ of length k, if $(\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$ for some $st_i, st_{i+k} \subseteq S$, then $\widetilde{\pi}$ has a corresponding concrete path which begins at a state in st_i and ends in a state in st_{i+k}.

- Consider that $\widetilde{\pi}' = \widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}$ is a path of length $k+1$ where $(\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_t (\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$. By induction hypothesis, there exists a concrete path that begins at some state $s_i \in st_i$ and ends in $s_{i+k} \in st_{i+k}$. By the same analysis as in the proof of proposition 5.3, there exists $s_{i-1} \in st_{i-1}$ and $\alpha_i \in h_A^{-1}(\widetilde{\alpha}_i)$ such that $s_{i-1} \xrightarrow{\alpha_i} s_i$. Hence, there exists a concrete path from s_{i-1} to s_{i+k}.

- Consider that $\widetilde{\pi}' = \widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}$ is a path of length $k+1$ where $(\widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_e (\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$. By induction hypothesis, there exists a concrete path that begins at some state $s_i \in st_i$ and ends in $s_{i+k} \in st_{i+k}$. By the definition of $\Rightarrow_e$ and proposition 5.3, s_i is reachable from some initial states in the concrete model, which is a requirement by definition 5.15. From $l_a(s_i) \in L_a(st_{i-1})$ we conclude that there exists $s_{i-1} \in st_{i-1}$ such that $l_a(s_i) = l_a(s_{i-1})$. Hence we select $s_{i-1} \sim_a s_i$ as the corresponding epistemic transition in the concrete model. Therefore, there exists a concrete path from s_{i-1} to s_{i+k}.

□

In the case that $\widetilde{\pi} = \widetilde{s}_0 \to \cdots \to \widetilde{s}_n$ is a path in the counterexample and $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}_n, st_n)$, then there exists a corresponding concrete path beginning at some initial state $s_0 \in S_0 \cap h^{-1}(\widetilde{s}_0)$ which ends at some state $s_n \in st_n$.

Proposition 5.5 (Completeness of $\Rightarrow^*$)**.** Let $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ be a path in the abstract model. If there exists a concrete path $\pi = s_1 \to \cdots \to s_n$ corresponding to $\widetilde{\pi}$ and $s_1 \in st_1 \subseteq h^{-1}(\widetilde{s}_1)$, then $(\widetilde{\pi}, st_1) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$.

Proof. For the completeness proof, we use induction over the length of the counterexamples.

Base case: $\widetilde{\pi} = \widetilde{s}_1$ and $\pi = s_1$. Then we will have no transition and the proposition automatically holds.

Inductive case: Assume by inductive hypothesis that for all $\widetilde{\pi} = \widetilde{s}_i \to \cdots \to \widetilde{s}_{i+k}$ of length k, if there exists a path $\pi = s_i \to \cdots \to s_{i+k}$ which corresponds to $\widetilde{\pi}$ and $s_i \in st_i \subseteq h^{-1}(\widetilde{s}_i)$, then $(\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$ for some $\emptyset \subset st_{i+k} \subseteq S$.

- Consider that $\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}$ is a path of length $k+1$ which has the corresponding concrete path $s_{i-1} \xrightarrow{\alpha_i} s_i \,||\, \pi$. Let $st_{i-1} \in h^{-1}(\widetilde{s}_{i-1})$ be a set of states where $s_{i-1} \in st_{i-1}$. Then the transition $(\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_t (\widetilde{\pi}, st_i)$ leads to the set st_i as the successors of the states in st_{i-1} with respect to the actions in $h_A^{-1}(\widetilde{\alpha}_i)$. As $\alpha_i \in h_A^{-1}(\widetilde{\alpha}_i)$, we have $s_i \in st_i$. Therefore by inductive hypothesis, we have $(\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_t (\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$ or equivalently $(\widetilde{s}_{i-1} \xrightarrow{\widetilde{\alpha}_i} \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$.

- Consider that $\widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}$ is a path of length $k+1$ which has the corresponding concrete path $s_{i-1} \sim_a s_i \,||\, \pi$. Let $st_{i-1} \in h^{-1}(\widetilde{s}_{i-1})$ be a set of states where $s_{i-1} \in st_{i-1}$. Then the transition $(\widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_e (\widetilde{\pi}, st_i)$ leads to the set st_i which contains the reachable states with the same local states as the states in st_{i-1}. Therefore, $s_i \in st_i$ and by inductive hypothesis we have $(\widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow_e (\widetilde{\pi}, st_i) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$ or equivalently $(\widetilde{s}_{i-1} \sim_a \widetilde{s}_i \,||\, \widetilde{\pi}, st_{i-1}) \Rightarrow^* (\widetilde{s}_{i+k}, st_{i+k})$.

□

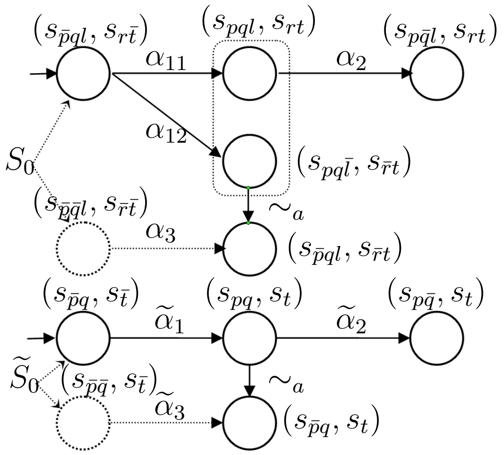

Figure 5.4: The transition system on the top is the concrete model and on the bottom is the abstract one obtained by making the propositions l and r invisible.

Forward transition rules in figure 5.3 are sufficient to check *linear counterexamples* or equivalently, paths. To extend the counterexample checking to tree-like counterexample, extra procedures are required. We show the problem in the following example:

Example 5.3. Figure 5.4 demonstrates the transition system for a concrete interpreted system on top, and the abstract system on the bottom. The model contains two agents, e as the environment and a as regular agent. States are shown as tuples where the first element is the local state of e and the second is the local state of a. The diagram distinguishes the states by using the value of local propositions as the subscript. The abstract model is generated by making the local proposition l of environment and r of agent a invisible.

We aim to verify $AG(p \to (K_a p \vee AGq))$ over the concrete model. This property holds for the original model, while it does not hold for the abstract one. The counterexample generated is:

$$\widetilde{ce} = \{(s_{\bar{p}q}, s_{\bar{t}}) \xrightarrow{\widetilde{\alpha}_1} (s_{pq}, s_t) \xrightarrow{\widetilde{\alpha}_2} (s_{p\bar{q}}, s_t), (s_{\bar{p}q}, s_{\bar{t}}) \xrightarrow{\widetilde{\alpha}_1} (s_{pq}, s_t) \sim_a (s_{\bar{p}q}, s_t)\}$$

To find out if there exists any concrete counterexample that corresponds to $\widetilde{ce}$, we check the paths in $\widetilde{ce}$ one by one. We show the paths in $\widetilde{ce}$ by $\widetilde{\pi}_1$ and $\widetilde{\pi}_2$. The paths $\widetilde{\pi}_1$ and $\widetilde{\pi}_2$ correspond to the concrete paths $\pi_1 = (s_{\bar{p}ql}, s_{r\bar{t}}) \xrightarrow{\alpha_{11}} (s_{pql}, s_{rt}) \xrightarrow{\alpha_2} (s_{p\bar{q}l}, s_{rt})$ and $\pi_2 = (s_{\bar{p}ql}, s_{r\bar{t}}) \xrightarrow{\alpha_{12}} (s_{pq\bar{l}}, s_{\bar{r}t}) \sim_a (s_{\bar{p}ql}, s_{\bar{r}t})$. Although all the paths in the counterexample have corresponding concrete paths, the tree does not correspond to a concrete tree. This is because if we select (s_{pql}, s_{rt}) as the corresponding state for (s_{pq}, s_t), then the leaf $(s_{\bar{p}ql}, s_{\bar{r}t})$ is not reachable from it. A similar situation happens when we select $(s_{pq\bar{l}}, s_{\bar{r}t})$. Therefore, the tree-like counterexample is spurious.

To verify a tree-like counterexample, we introduce two transition rules BackwardTCheck

$$\textsc{BackwardTCheck}\ \frac{\begin{array}{c}(\pi, S_0 \cap h^{-1}(\mathbf{Root}(\pi))) \Rightarrow^* (\widetilde{s}, st') \\ h_A^{-1}(\widetilde{\alpha}) = \{\alpha_1, \dots, \alpha_n\} \qquad rs = \bigcup_{i=1}^{n} \Theta_{\alpha_i}^{-1}(st) \cap st'\end{array}}{(\pi \,||\, \widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}', st) \Leftarrow_t (\pi, rs) \qquad r_{\widetilde{s}} := rs}$$

$$\textsc{BackwardECheck}\ \frac{\begin{array}{c}(\pi, S_0 \cap h^{-1}(\mathbf{Root}(\pi))) \Rightarrow^* (\widetilde{s}, st'') \\ \pi' = \widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \dots \xrightarrow{\widetilde{\alpha}'_m} \widetilde{s}' \text{ is the temporal path to } \widetilde{s}' \text{ where } \widetilde{s}'_0 \in \widetilde{S}_0 \\ (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow^* (\widetilde{s}', st') \\ \widehat{st} = \{s \in st'' \mid l_a(s) \in L_a(st \cap st')\}\end{array}}{(\pi \,||\, \widetilde{s} \sim_a \widetilde{s}', st) \Leftarrow_e (\pi, \widehat{st}) \qquad r_{\widetilde{s}} := \widehat{st}}$$

Figure 5.5: Backward temporal and epistemic transition traversal. $\Theta_\alpha^{-1}(st)$ computes the set of predecessors of the states in st with respect to the transitions made by action α.

and $\textsc{BackwardECheck}$ denoted by $\Leftarrow_t$ and $\Leftarrow_e$. The transition rules find all the predecessors of the states in st (figure 5.5) with respect to the temporal or epistemic transitions in a backward manner which reside in the set of reachable states through the path. We write $\Leftarrow^*$ to denote a sequence of backward transitions $\Leftarrow_t$ and $\Leftarrow_e$.

Assume that $\widetilde{\pi} = \widetilde{s}_0 \to \cdots \to \widetilde{s}_n$ is a path in the counterexample $\widetilde{ce}$ which $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$. st_n contains all the states in the leaves of the concrete paths corresponding to $\widetilde{\pi}$. The point is not all the concrete states that are traveresed in $\Rightarrow^*$ can reach the states in st_n. If $\widetilde{s} \in \mathbf{Vert}(\widetilde{\pi})$, then $(\widetilde{\pi}, st_n) \Leftarrow^* (\widetilde{s}_0, st_0)$ finds the set of states $r_{\widetilde{s}}$ which contains the reachable states in $h^{-1}(\widetilde{s})$ that lead to some states in st_n along the concrete paths corresponding to $\widetilde{\pi}$. st_0 contains the initial states that lead to the states in st_n. We use the notation $r^{\widetilde{\pi}}_{\widetilde{s}}$ to relate $r_{\widetilde{s}}$ with the path $\widetilde{\pi}$. Note that to find $r^{\widetilde{\pi}}_{\widetilde{s}}$, we first need to find st_n through $\Rightarrow^*$ transition.

Assume that $\widetilde{\Pi} \subseteq \widetilde{ce}$. If $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ then we define $r^{\widetilde{\Pi}}_{\widetilde{s}} = \cap_{\widetilde{\pi} \in \widetilde{\Pi}} r^{\widetilde{\pi}}_{\widetilde{s}}$. If $\widetilde{s} \notin \mathbf{Vert}(\widetilde{\pi})$, then we stipulate $r^{\widetilde{\pi}}_{\widetilde{s}} = h^{-1}(\widetilde{s})$. We also stipulate $r^{\emptyset}_{\widetilde{s}_0} = S_0 \cap h^{-1}(\widetilde{s}_0)$ where $\widetilde{s}_0 = \mathbf{Root}(\widetilde{ce})$ and $r^{\emptyset}_{\widetilde{s}} = h^{-1}(\widetilde{s})$ for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ where $\widetilde{s} \neq \widetilde{s}_0$.

Proposition 5.6 (Soundness of counterexample checking)**.** A counterexample $\widetilde{ce}$ in the abstract model has a corresponding concrete one if:

1. for each path $\widetilde{\pi} \in \widetilde{ce}$, there exists $\emptyset \subset st \subseteq S$ such that $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}', st)$ where $\widetilde{s}_0 = \mathbf{Root}(\widetilde{ce})$ and $\widetilde{\pi}$ ends in $\widetilde{s}'$.

2. for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce}) : r^{\widetilde{ce}}_{\widetilde{s}} \neq \emptyset$.

Proof. By the soundness of $\Rightarrow^*$, all the paths in $\widetilde{\pi}$ correspond to some concrete paths which satisfy the requirements in the definitions 5.15 and 5.16. Now for each $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$, we pick a state $s \in r_{\widetilde{s}}^{\widetilde{ce}}$ as the corresponding state. For each path in $\widetilde{ce}$ and between all the corresponding concrete paths, we pick the one which contains the selected states as its vertices. The union of the selected paths builds a concrete counterexample that satisfies the requirements in definition 5.16. □

Procedure 4 Counterexample checking algorithm

```
function CheckCE($\widetilde{ce}, I, h$)
    ▷ Input: $\widetilde{ce}$ is the counterexample, $I$ is the concrete model and $h$ is the abstraction function
    ▷ Output: returns true if a concrete counterexample exists. Returns false otherwise.
    $\{\widetilde{s}_0, \ldots, \widetilde{s}_n\} = \mathbf{Vert}(\widetilde{ce})$                    ▷ $\widetilde{s}_0 = \mathbf{Root}(\widetilde{ce})$
    $\widetilde{\Pi} = \emptyset$
    $r_{\widetilde{s}_0}^{\widetilde{\Pi}} = S_0 \cap h^{-1}(\widetilde{s}_0), r_{\widetilde{s}_1}^{\widetilde{\Pi}} = h^{-1}(\widetilde{s}_1), \ldots, r_{\widetilde{s}_n}^{\widetilde{\Pi}} = h^{-1}(\widetilde{s}_n)$
    for all $\widetilde{\pi} \in \widetilde{ce}$ do
        if $(\widetilde{\pi}, r_{\widetilde{s}_0}^{\widetilde{\Pi}}) \Rightarrow^* (\widetilde{s}', st)$ and $st \neq \emptyset$ then          ▷ $\widetilde{\pi}$ ends at the state $\widetilde{s}'$
            ▷ there exists some concrete path corresponding to $\widetilde{\pi}$
            for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ do
                determine $\hat{r}_{\widetilde{s}}^{\widetilde{\pi}}$ from $(\widetilde{\pi}, st) \Leftarrow^* (\widetilde{s}_0, st')$
                ▷ determine the concrete states corresponding to $\widetilde{s}$
                $r_{\widetilde{s}}^{\widetilde{\Pi}\cup\{\widetilde{\pi}\}} := r_{\widetilde{s}}^{\widetilde{\Pi}} \cap r_{\widetilde{s}}^{\widetilde{\pi}}$
                if $r_{\widetilde{s}}^{\widetilde{\Pi}\cup\{\widetilde{\pi}\}} = \emptyset$ then
                    ▷ no common concrete state for $\widetilde{s}$ between concrete paths exists
                    return false
                end if
            end for
            $\widetilde{\Pi} := \widetilde{\Pi} \cup \{\widetilde{\pi}\}$
        else
            return false
        end if
    end for
    return true
end function
```

Proposition 5.7 (Completeness of counterexample checking)**.** Assume that $\widetilde{ce}$ corresponds to a concrete counterexample ce. Then both the items 1 and 2 in proposition 5.6 hold.

Proof. By definition 5.16, there is a one-to-one correspondence between the paths of the two counterexamples. By completeness of $\Rightarrow^*$, item 1 holds for all the paths in $\widetilde{ce}$. Now Assume that $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ and s is the corresponding state in ce. Then for all $\widetilde{\pi} \in \widetilde{ce}$, we have $s \in r_{\widetilde{s}}^{\widetilde{\pi}}$, and therefore $s \in r_{\widetilde{s}}^{\widetilde{ce}}$. Hence we have $r_{\widetilde{s}}^{\widetilde{ce}} \neq \emptyset$, as required for item 2. □

Procedure 4 expresses the tree-like counterexample checking method in a more refined manner. CHECKCE iterates over the paths in $\widetilde{ce}$ and checks if they corresponds to some paths in the concrete model by using proposition 5.4 and the transition rule $\Rightarrow^*$. If $\widetilde{\pi}$ corresponds to some concrete paths, then for each state $\widetilde{s}$ in $\widetilde{\pi}$, the algorithm finds all the concrete states $r^{\widetilde{\pi}}_{\widetilde{s}}$ in $h^{-1}(\widetilde{s})$ that lead to the leaf states of the concrete paths by applying $\Leftarrow^*$ over $\widetilde{\pi}$. In each loop iteration, $\widetilde{\Pi}$ stores the paths in $\widetilde{ce}$ that are processed in previous iterations. The set $r^{\widetilde{\Pi}}_{\widetilde{s}}$ stores the concrete states that are common between the paths in $\widetilde{\Pi}$ and should remain non-empty during the process of counterexample checking. The procedure returns `false` if no corresponding tree-like counterexample for $\widetilde{ce}$ exists. Otherwise it returns `true`.

Example 5.4. We recall the transition system in example 5.3. As also discovered in the example, the paths $\widetilde{\pi}_1$ and $\widetilde{\pi}_2$ correspond to the concrete paths $\pi_1 = (s_{\bar{p}ql}, s_{r\bar{t}}) \xrightarrow{\alpha_{11}} (s_{pql}, s_{rt}) \xrightarrow{\alpha_2} (s_{p\bar{q}l}, s_{rt})$ and $\pi_2 = (s_{\bar{p}ql}, s_{r\bar{t}}) \xrightarrow{\alpha_{12}} (s_{pq\bar{l}}, s_{\bar{r}t}) \sim_a (s_{\bar{p}ql}, s_{\bar{r}t})$. By backward traversing through the first path and for the states in $h^{-1}((s_{pq}, s_t))$, we find that only the state (s_{pql}, s_{rt}) leads to the final state on π_1 and so, $r^{\widetilde{\pi}_1}_{(s_{pq},s_t)} = \{(s_{pql}, s_{rt})\}$. The same approach for π_2 results in $r^{\widetilde{\pi}_2}_{(s_{pq},s_t)} = \{(s_{pq\bar{l}}, s_{\bar{r}t})\}$. As $r^{\widetilde{\pi}_1}_{(s_{pq},s_t)} \cap r^{\widetilde{\pi}_2}_{(s_{pq},s_t)} = \emptyset$, the state (s_{pq}, s_t) can not be assigned to a concrete single state. Therefore, $\widetilde{ce}$ is spurious.

5.5.3 Refinement of the abstraction

If the counterexample is found to be spurious, then the abstraction should be refined. The abstract model is generated by making some propositions in the concrete model invisible. For the refinement, we split some states in the abstract model by putting some of the invisible propositions back into the model. These propositions should be selected in such a way that when verifying the refined model, the same counterexample does not appear again. In this section, we provide the mechanism for refining the abstraction.

Let $\widetilde{ce}$ be a spurious counterexample. We define two transition rules TEMPORALTREE which is denoted by $\Rightarrow^{\widetilde{\Pi}}_t$ and EPISTEMICTREE denoted by $\Rightarrow^{\widetilde{\Pi}}_e$ where $\widetilde{\Pi} \subseteq \widetilde{ce}$ in figure 5.6. As before, $\Rightarrow^{\widetilde{\Pi}}_*$ denotes a sequence of temporal and epistemic transitions of the type $\Rightarrow^{\widetilde{\Pi}}_t$ and $\Rightarrow^{\widetilde{\Pi}}_e$. We use the following technique in order to find the state in the spurious counterexample which needs to be split:

The state $\widetilde{s}_i \in \mathbf{Vert}(\widetilde{ce})$ is a *failure state* if there exists $\widetilde{\Pi} \subseteq \widetilde{ce}$ and $\widetilde{\pi} \in \widetilde{ce} \backslash \widetilde{\Pi}$ such that:

1. For all $\widetilde{s} \in \mathbf{Vert}(\widetilde{\Pi}) : r^{\widetilde{\Pi}}_{\widetilde{s}} \neq \emptyset$

2. $\widetilde{\pi} = \widetilde{\pi}_1 \mid\mid \widetilde{s}_i(\xrightarrow{\widetilde{\alpha}_{i+1}} \mid \sim_a)\widetilde{s}_{i+1} \mid\mid \widetilde{\pi}_2$ such that $(\widetilde{\pi}, r^{\widetilde{\Pi}}_{\widetilde{s}_0}) \Rightarrow^{\widetilde{\Pi}}_* (\widetilde{\pi}_1, st_d) \Rightarrow^{\widetilde{\Pi}}_{(t|e)} (\widetilde{\pi}_2, \emptyset)$ for some $st_d \neq \emptyset$.

$$\textsc{TemporalTree}\ \frac{h_A^{-1}(\widetilde{\alpha}) = \{\alpha_1, \dots, \alpha_n\}}{(\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}' \,||\, \pi, st) \Rightarrow_t^{\widetilde{\Pi}} (\pi, \bigcup\limits_{i=1}^{n} \Theta_{\alpha_i}(st) \cap r_{\widetilde{s}'}^{\widetilde{\Pi}})}$$

$$\textsc{EpistemicTree}\ \frac{\begin{array}{c}\pi' = \widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \dots \xrightarrow{\widetilde{\alpha}'_m} \widetilde{s}' \text{ is a temporal path to } \widetilde{s}' \text{ where } \widetilde{s}'_0 \in \widetilde{S}_0 \\ (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^{*} (\widetilde{s}', st') \qquad \hat{st} = \{s \in st' \cap r_{\widetilde{s}'}^{\widetilde{\Pi}} \mid l_a(s) \in L_a(st)\}\end{array}}{(\widetilde{s} \sim_a \widetilde{s}' \,||\, \pi, st) \Rightarrow_e^{\widetilde{\Pi}} (\pi, \hat{st})}$$

Figure 5.6: Transition rules for finding failure state in a tree-like counterexample.

For a spurious counterexample, such $\widetilde{\Pi}$ and $\widetilde{\pi}$ exists. Otherwise, we will have $r_{\widetilde{s}}^{\widetilde{ce}} \neq \emptyset$ for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$, which contradicts proposition 5.6.

Based on Item 1), the sub-tree $\widetilde{\Pi}$ has a corresponding counterexample in the concrete model. In item 2), $\widetilde{\pi}$ traverses over the concrete states that belong to the set of concrete trees corresponding to $\widetilde{\Pi}$ and gets to the set of states $st_d \subseteq h^{-1}(\widetilde{s}_i)$ with no transition to a state in $r_{\widetilde{s}_{i+1}}^{\widetilde{\Pi}}$. In the standard terminology as in [30], $\widetilde{s}_i$ is called *failure state*. We use the term *dead end state* for the states in st_d which the concrete paths end up with and can not go further. *Bad states* are the states in $h^{-1}(\widetilde{s}_i)$ that have transition to some states in $r_{\widetilde{s}_{i+1}}^{\widetilde{\Pi}}$. Note that in a path counterexample, we have that $r_{\widetilde{s}_{i+1}}^{\widetilde{\Pi}} = h^{-1}(\widetilde{s}_{i+1})$.

The process of finding a failure state in the counterexample $\widetilde{ce}$ proceeds as follows:

1. Set $\widetilde{\Pi}$ to empty set at the beginning

2. Find $r_{\widetilde{s}}^{\widetilde{\Pi}}$ for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ (as also mentioned in section 5.5.2, $r_{\widetilde{s}_0}^{\emptyset} = S_0 \cap h^{-1}(\widetilde{s}_0)$ where $\widetilde{s}_0 = \mathbf{Root}(\widetilde{ce})$ and $r_{\widetilde{s}}^{\emptyset} = h^{-1}(\widetilde{s})$ for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ where $\widetilde{s} \neq \widetilde{s}_0$)

3. Pick a path $\widetilde{\pi} \in \widetilde{ce}$ that does not exist in $\widetilde{\Pi}$

4. Apply $\Rightarrow_*^{\widetilde{\Pi}}$ over $(\widetilde{\pi}, r_{\widetilde{s}_0}^{\widetilde{\Pi}})$ to find failure state. If a failure state exists over $\widetilde{\pi}$, then exit and refine the model

5. Add $\widetilde{\pi}$ to $\widetilde{\Pi}$ and return to step 2. Note that we are considering that the counterexample is found to be spurious (by the procedure 4) and therefore, such failure state will be found before all the paths in $\widetilde{ce}$ are added to $\widetilde{\Pi}$.

For the implementation, the above process can be easily incorporated into the procedure 4.

To refine the model, we find the propositions that having them invisible results in generating spurious counterexample. First assume that the transition from $\widetilde{s}_i$ to $\widetilde{s}_{i+1}$ is temporal, say $\widetilde{s}_i \xrightarrow{\widetilde{\alpha}_{i+1}} \widetilde{s}_{i+1}$. Two situations can result in a transition of type $\Rightarrow_t^{\widetilde{\Pi}}$ from st_d to an empty set of states:

- There exists no $\alpha_{i+1} \in h^{-1}(\widetilde{\alpha}_{i+1})$ such that $\Theta_{\alpha_{i+1}}(st_d) \neq \emptyset$. Therefore, no action has the permission to be performed on the states of st_d. Assume that ϕ_d is the formula that represents the set of states st_d. As the state space is finite, the formula representing the states always exists. Therefore, for all $\alpha_{i+1} \in h_A^{-1}(\widetilde{\alpha}_{i+1})$ with ℓ_{i+1} as the permission, we have $\phi_d \wedge \ell_{i+1} \equiv \bot$. We call ℓ_{i+1} *conflict formula* and ϕ_d *base formula.*

- For some $\alpha_{i+1} \in h^{-1}(\widetilde{\alpha}_{i+1})$ we have $\Theta_{\alpha_{i+1}}(st_d) \neq \emptyset$. By the definition of $\Rightarrow_t$ we have $\Theta_{\alpha_{i+1}}(st_d) \cap r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}} = \emptyset$ where $r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}} \neq \emptyset$. If ϕ is the formula representing $\Theta_{\alpha_{i+1}}(st_d)$ and ψ the formula representing $r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}}$, then we have $\psi \wedge \phi \equiv \bot$. We call ϕ conflict formula and ψ base formula.

The other situation is when the transition $\widetilde{s}_i$ and $\widetilde{s}_{i+1}$ is epistemic, say $\widetilde{s}_i \sim_a \widetilde{s}_{i+1}$. Three situations can result in the epistemic transition $\Rightarrow_e^{\widetilde{\Pi}}$ to an empty set of states:

- π' as the witness of the reachability of $\widetilde{s}_{i+1}$ in $\Rightarrow_e^{\widetilde{\Pi}}$ is spurious. Then the refinement should be guided by analysing π' instead of the main spurious path.

- Suppose that π' has corresponding concrete paths, i.e. $(\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^* (\widetilde{s}_{i+1}, st')$ where $st' \neq \emptyset$. By the definition of $\Rightarrow_e$, the epistemic transition results in an empty set of states if $st' \cap r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}} = \emptyset$. If ϕ is the formula representing st' and ψ the formula representing $r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}}$, then we call ϕ conflict formula and ψ base formula.

- The third reason for the epistemic transition to an empty set is when no shared *local state* exists between the states of st_d and $st' \cap r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}}$ where st' is the set of reachable states according to the previous item and both the sets are non-empty. In the other words, $L_a(st_d) \cap L_a(st' \cap r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}}) = \emptyset$. The formula representing the local states in st_d with respect to the agent a is called base formula, and the formula representing the local states of $st' \cap r^{\widetilde{\Pi}}_{\widetilde{s}'_{i+1}}$ is the conflict formula.

To refine the model, we return some hidden propositions to separate the set of dead end states from the rest of the states. This can simply be done by adding all the propositions occurring in *conflict clauses* to the abstract model.

Definition 5.18. (conflict clause) Let ϕ be the base formula and ψ the conflict formula. Let $\mathbf{cnf}(\psi)$ denote the set containing all the conjuncts appear in conjunctive normal form of ψ. Then $c \in \mathbf{cnf}(\psi)$ is a *conflict clause* if $c \wedge \phi \equiv \bot$.

If the propositions that occur in one of the conflict clauses become visible, then the spurious strategy will not happen in the refined model again. In the case of temporal transition, we add the propositions in the conflict clauses for *all the conflicting actions*. To have the smallest possible refinement, we should look for the conflict classes with the *smallest number* of literals.

5.5.4 An example of student information system

We illustrate our abstraction refinement method by the example of *student information system* (SIS) presented in chapter 4. In this section, the approach for finding conflict clauses and the refinement of the abstract model is demonstrated using the same temporal property as in query 4.5. This example is interesting for us as it is also used in [94] and [11] to demonstrate the verification time and memory usage in their verification approach. The refinement method for the case of epistemic properties similarly follows the instructions in section 5.5.3.

Let $\Sigma = \Sigma_{Ag} = \{\mathtt{a}_1, \ldots, \mathtt{a}_5\}$. We use the convention that the first parameter of an action is the agent that performs it. The action rules in SIS simple policy are as follows:

$\mathtt{assignDemonstrator}(u, d, s)$:
$\quad\{\mathtt{+demonstratorOf}(d, s)\} \leftarrow \mathtt{lecturer}(u) \wedge \mathtt{higher}(d, s) \wedge \neg\mathtt{higher}(s, d)$
$\mathtt{resignAsDemonstrator}(u, s)$:
$\quad\{\mathtt{-demonstratorOf}(u, s)\} \leftarrow \mathtt{demonstratorOf}(u, s)$

The first action rule states that if u is a lecturer and d is in higher level than s, then u can assign d as the demonstrator of s. The second one stipulates that if u is the demonstrator of s, then u can resign as the demonstrator.

The policy $\mathcal{C}$ is the set of actions derived by instantiating the above rules with the objects in Σ_{Ag}. Assume that at the beginning, the agents $\mathtt{a}_1$ and $\mathtt{a}_2$ are not the demonstrator of each other. Then $I_{\mathcal{C}}$ is the interpreted system derived from $\mathcal{C}$ where

$$(I_{\mathcal{C}}, s_0) \models \neg\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3) \wedge \neg\mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)$$

The safety property we are interested in verifying is “is it not possible to get into a state where $\mathtt{a}_2$ and $\mathtt{a}_3$ are allocated as the demonstrators of each other?”, represented by the CTL formula $AG(\neg(\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3) \wedge \mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)))$.

For the initial abstraction, we only keep the propositions $\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3)$ and $\mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)$ in the system. Substituting the parameters in the rules with the appropriate objects creates the set of actions. For instance and in the following action in $(I_{\mathcal{C}}$, agent $\mathtt{a}_1$ assigns $\mathtt{a}_2$ as the demonstrator of $\mathtt{a}_3$:

$$\begin{aligned}&\texttt{assignDemonstrator}(\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3)\texttt{ :}\\ &\qquad \{\texttt{+demonstratorOf}(\mathtt{a}_2,\mathtt{a}_3)\} \leftarrow \texttt{lecturer}(\mathtt{a}_1)\ \wedge\texttt{higher}(\mathtt{a}_2,\mathtt{a}_3) \wedge \neg\texttt{higher}(\mathtt{a}_3,\mathtt{a}_2)\end{aligned}$$

Hiding the propositions $\texttt{lecturer}(\mathtt{a}_1)$, $\texttt{higher}(\mathtt{a}_2,\mathtt{a}_3)$ and $\texttt{higher}(\mathtt{a}_3,\mathtt{a}_2)$ will turn the evolution rule into $\{\texttt{+demonstratorOf}(\mathtt{a}_2,\mathtt{a}_3)\}\leftarrow \top$.

As the initial abstraction hides $\texttt{lecturer}(\mathtt{a}_1)$, ..., $\texttt{lecturer}(\mathtt{a}_5)$, then all the actions $\texttt{assignDemonstrator}(\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3),\ldots,\texttt{assignDemonstrator}(\mathtt{a}_5,\mathtt{a}_2,\mathtt{a}_3)$ have the same permission and the same effect, and therefore they belong to the same equivalence class. Now we build up the abstract model using the rules in section 5.3 and verify the abstract model. The verification produces the following counterexample:

$$\widetilde{ce} = \{\ \ \widetilde{s}_0 \xrightarrow{[\texttt{assignDemonstrator}(\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3)]} \widetilde{s}_1 \xrightarrow{[\texttt{assignDemonstrator}(\mathtt{a}_1,\mathtt{a}_3,\mathtt{a}_2)]} \widetilde{s}_3\}$$

where

$$\begin{aligned}(\widetilde{I_\mathcal{C}},\widetilde{s}_0) &\models \neg\texttt{demonstratorOf}(\mathtt{a}_2,\mathtt{a}_3) \wedge \neg\texttt{demonstratorOf}(\mathtt{a}_3,\mathtt{a}_2)\\ (\widetilde{I_\mathcal{C}},\widetilde{s}_1) &\models \texttt{demonstratorOf}(\mathtt{a}_2,\mathtt{a}_3) \wedge \neg\texttt{demonstratorOf}(\mathtt{a}_3,\mathtt{a}_2)\\ (\widetilde{I_\mathcal{C}},\widetilde{s}_2) &\models \texttt{demonstratorOf}(\mathtt{a}_2,\mathtt{a}_3) \wedge \texttt{demonstratorOf}(\mathtt{a}_3,\mathtt{a}_2)\end{aligned}$$

The first transition in $\widetilde{ce}$ has five corresponding transition as the result of performing the concrete actions in $[\texttt{assignDemonstrator}(\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3)]$. For the next step and from the successor set of states, none of the actions in the equivalence class of $\texttt{assignDemonstrator}$ $(\mathtt{a}_1,\mathtt{a}_3,\mathtt{a}_2)$ can be performed. We can see that the conflict clause $\texttt{higher}(\mathtt{a}_3,\mathtt{a}_2)$ is common between all the actions in that equivalence class. Addition of the proposition $\texttt{higher}(\mathtt{a}_3,\mathtt{a}_2)$ prevents the same counterexample to occur in the refined model. Only one refinement step is sufficient to show that the safety property is true in the abstract model and hence, is true in the concrete one.

5.5.5 Going beyond ACTLK

While this section develops a fully automated abstraction refinement method for the verification of temporal-epistemic properties that reside the category of ACTLK over an access control system which is modelled by an interpreted system, some important epistemic safety properties does not reside in this category. For instance and in a conference paper review system, it is valuable for policy designers to verify that for all reachable states, an author of a paper cannot find out ($\neg K$) who is the reviewer of his own paper (see the first property in example 5.2). Although we are able to verify such properties in the concrete model, we cannot apply automated counterexample-guided abstraction and refinement for such properties.

Let us explore the problem. Assume that for the abstract system $\widetilde{I}$, abstract state $\widetilde{s}$ and agent a, $(\widetilde{I}, \widetilde{s}) \models \neg K_a \varphi$. That means there exists a state $\widetilde{s'}$ such that $\widetilde{s'} \sim_a \widetilde{s}$ and $(\widetilde{I}, \widetilde{s'}) \models \neg\varphi$. If s is a state in the concrete model where $h(s) = \widetilde{s}$, then the satisfaction relation $(\widetilde{I}, \widetilde{s}) \models \neg K_a \varphi$ implies $(I, s) \models \neg K_a \varphi$ if it guarantees the existence of a *reachable* state $s' \in h^{-1}(\widetilde{s'})$ such that $s' \sim_a s$ and $(I, s') \models \neg\varphi$.

First of all, if such s' exists, the satisfaction relation $(\widetilde{I}, \widetilde{s'}) \models \neg\varphi$ still does not imply $(I, s') \models \neg\varphi$ when φ is ACTLK except if φ is simply a propositional formula which is the case for many of the properties that we are interested in. Second, the relation $\widetilde{s'} \sim_a \widetilde{s}$ in the abstract model does not imply $s' \sim_a s$ in the concrete model for some reachable state $s' \in h^{-1}(\widetilde{s'})$. In the case that $(\widetilde{I}, \widetilde{s'}) \not\models \neg\varphi$, the model-checker produces a counterexample that can be checked using the method that is developed in this section and then the abstract model can be refined. In the case that the satisfaction relation holds, the model-checker does not produce any witness.

To complete our work for the properties that deal with the negation of knowledge operator, we restrict the formula in scope of the knowledge operators to propositional formulas. Then we use an interactive refinement procedure in the following way: we abstract the interpreted system in the standard way that we described. If the property does not hold in the abstract model, the counterexample will be checked in the concrete model and the abstract model will be refined if it is required. If the property turned to be true in the abstract model as a result of the satisfaction of $\neg K_a$ (for which there is no witness in the abstract model), then we refine the local state of the agent a in an interactive manner. In this way, the tool asks the user to selects a set of invisible *local propositions* to be added in the next round if required. This process will continue until a valid counterexample is found, or the local state becomes concretized. In the case that the safety property does not hold in the concrete model (where information leakage vulnerability exists), then there is a chance to find it out with the abstract model when the local states are still abstract.

Query 5.6 in section 5.6 demonstrates such an approach. Our experimental results show that while the verification time in interactive approach depends on the choices of the user, in general, the whole verification process is still much lower than verifying the concrete model at the beginning. Moreover, this interactive approach can turn to a fully automated way if the tool can use a heuristics based on the policy which can identify the hidden propositions with higher dependencies to the property and add them automatically to the abstract model for the refinement.

In our implementation and in the cases when the knowledge is the result of reading system information and not reasoning, it is sufficient to keep only the local state propositions which support the readability of that information. Therefore, the local state will remain in the least possible abstract form and is not required to be refined when the property does not hold. Query 5.3 in section 5.6 demonstrates such a property.

5.6 Case studies and experimental results

The policy verification method is implemented in Microsoft F#. The implementation uses MCMAS as the model checker for interpreted systems. A policy written in the language proposed in chapter 3 will be translated into ISPL (MCMAS script language) and verified. In the case of applying abstraction and refinement, an abstract interpreted system is generated in ISPL and then, MCMAS is invoked to verify the model. If MCMAS produces a counterexample as the witness of a failing property, the counterexample is checked to have a corresponding one in the concrete model. If it is spurious, a refined abstract model will be generated automatically. To find conflict clauses for the refinement, we use Microsoft Z3 SMT Solver [39].

We use the query of the form *init* : φ where *init* is the formula representing the initial states and φ is the property we aim to verify. We compare runtime and memory usage for different case studies and queries with and without applying abstraction. We also compare the results with other verification methods (RW and PoliVer) when the property allows us.

5.6.1 Case study: a student information system (SIS)

We recall the *student information system* policy introduced in section 5.5.4 as the case study. For such a system, it is important to ensure that no two students can be assigned as each other's demonstrator. If $\mathtt{a}_2, \mathtt{a}_3 \in \Sigma_{Ag}$, then query 5.1 checks the safety property that states no two students can be assigned as the demonstrator of each other.

Query 5.1.

$$\neg\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3) \wedge \neg\mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2) :$$
$$AG(\neg(\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3) \wedge \mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)))$$

As described in section 5.5.4, only the propositions $\mathtt{demonstratorOf}(\mathtt{a}_2, \mathtt{a}_3)$ and $\mathtt{demonstratorOf}(\mathtt{a}_3, \mathtt{a}_2)$ remain visible in the initial abstract model. Our method proves the correctness of the property for 10 agents in only one refinement round and by the addition of proposition $\mathtt{higher}(\mathtt{a}_3, \mathtt{a}_2)$ to the initial abstract model.

This query is similar to the query 6.8 in [94]. Becker [11] has compared the verification time between a planner, theorem prover and RW for this query for 10 agents and we will do the same comparison in experimental results section. As also mentioned in [11], it may not be a right comparison as the RW takes knowledge states into account and MCMAS is also somehow slower than conventional LTL/CTL model-checkers or planners. But still the comparison provides a view of how abstraction and refinement dramatically reduces the amount of memory and time in most of applications.

5.6.2 Case study: a conference paper review system (CRS)

We recall the *conference paper review system* example in chapter 4. Comparing to chapter 4, we apply the abstraction and refinement when verifying temporal and epistemic properties. In query 5.2, we verify the temporal property as in query 4.1 in the presence of abstraction refinement. Query 5.3 evaluates a non-ACTLK epistemic property using automated abstraction by approximating knowledge by the readability. This query is similar to the query 4.3 in the previous chapter. In section 5.6.4, we will verify several epistemic properties over the conference paper review system which can not be evaluated by PoliVer, RW or similar verification tools.

Assume that in the system, $\mathtt{a}_1 \in \Sigma_{Ag}$ and $\mathtt{p}_1 \in \Sigma$ is a paper. We aim to get sure that if a principal is the author of a paper, then it is not possible for him to be assigned as the reviewer of his own paper. Query 5.2 asks if such safety condition holds in CRS:

Query 5.2.

$$\mathtt{author}(\mathtt{p}_1, \mathtt{a}_1) \wedge \neg\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_1) : AG(\neg\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_1))$$

The initial abstract model contains only the proposition $\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_1)$. Our tool finds the satisfaction of the property in one refinement step and by making the proposition $\mathtt{author}(\mathtt{p}_1, \mathtt{a}_1)$ visible.

Another interesting query is the one that verifies if an agent has read the review of a paper, then it is not possible for him to submit a review for that paper later. Query 5.3 asks such question:

Query 5.3.

$$\begin{aligned}&\neg\mathtt{submittedreview}(\mathtt{p}_1, \mathtt{a}_1) \wedge \mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2) :\\ &AG(K_{\mathtt{a}_1}\mathtt{review}(\mathtt{p}_1, \mathtt{a}_2) \rightarrow AG(\neg\mathtt{submittedreview}(\mathtt{p}_1, \mathtt{a}_1))\end{aligned}$$

The above epistemic property is not ACTLK. But we are still able to use abstraction and refinement over the model. Firstly, knowledge about a review in the context of our system is derived by reading the review as a PC member knows the content of a review if he has already read the review. By this assumption, we only need to keep the supporting local propositions of agent $\mathtt{a}_1$ for reading $\mathtt{review}(\mathtt{p}_1, \mathtt{a}_2)$. Therefore, we do not need to apply abstraction over the local state of $\mathtt{a}_1$. Hence by the discussion in section 5.5.5, the automated abstraction and refinement method still works for the query 5.3.

This property does not hold in the system. For this query, the initial abstract model contains the propositions $\mathtt{review}(\mathtt{p}_1, \mathtt{a}_2)$ and $\mathtt{submittedreview}(\mathtt{p}_1, \mathtt{a}_1)$. The initial abstraction results in a spurious counterexample, and the verification in total contains five refinement steps (in our implementation). Finally, we will have the following counterexample, which also holds in the concrete model: $\mathtt{a}_1$ as a PC member first reads the review submitted by $\mathtt{a}_2$ when he is not the reviewer of $\mathtt{p}_1$. Then the chair ($\mathtt{a}_3$) allocates $\mathtt{a}_1$ as the reviewer of $\mathtt{p}_1$. At the end, $\mathtt{a}_1$ submits his review of the paper.

5.6.3 Case study: an employee information system (EIS)

We demonstrate the abstraction refinement technique when verifying temporal properties of *employee information system* presented in chapter 4.

Assume that in the system $\mathtt{a}_1 \in \Sigma_{Ag}$ and $\mathtt{a}_1$ is a manager. Query 5.4 checks if in the case that there is no director at the beginning, none of the agents can set a bonus for $\mathtt{a}_1$:

Query 5.4.

$$\bigwedge_i \neg\mathtt{director}(\mathtt{a}_i) \wedge \mathtt{manager}(\mathtt{a}_1) \wedge \neg\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1) : AG(\neg\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1))$$

The initial abstract model only contains the proposition $\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1)$ which appears in the query. The verification requires only one refinement round which makes the propositions $\mathtt{manager}(\mathtt{a}_1), \mathtt{director}(\mathtt{a}_1), \ldots, \mathtt{director}(\mathtt{a}_n)$ visible (n is the number of agents). The counterexample is: $\mathtt{a}_1$ resigns as a manager, and then $\mathtt{a}_2$, which is already a manager, allocates him the bonus $\mathtt{b}_1$.

The following query asks if it is impossible that some agent allocates a bonus to $\mathtt{a}_1$ and $\mathtt{a}_1$ remains a manager, assuming that no director exists in the system:

Query 5.5.

$$\bigwedge_i \neg\mathtt{director}(\mathtt{a}_i) \wedge \mathtt{manager}(\mathtt{a}_1) \wedge \neg\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1) : AG(\neg(\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1) \wedge \mathtt{manager}(\mathtt{a}_1)))$$

The refinement has only one round. At first, the propositions $\mathtt{bonus}(\mathtt{a}_1, \mathtt{b}_1)$ and $\mathtt{manager}(\mathtt{a}_1)$ remain visible. For the refinement, the propositions $\mathtt{director}(\mathtt{a}_1), \ldots,$

$\texttt{director}(\texttt{a}_n)$ will be added to the initial abstract model. This property does not hold in the system because if $\texttt{a}_1$ resigns as manager and $\texttt{a}_2$ allocates a bonus to him, then no agent in EIS is able to promote him again as a manager.

5.6.4 Case studies for reasoning about knowledge

We recall the conference paper review system in section 5.6.2 and add some extra rules to the policy. Let us add the following rules:

- All the PC members have access to the list of the papers assigned to other reviewers except the papers that they are the authors.
- The number of papers assigned to each reviewer is publicly available to all the PC members.

Query 5.6 investigates if it is possible for an agent, which is also an author of a paper in the conference to find out which reviewer is assigned to his paper after all the papers are assigned:

Query 5.6.

$$\texttt{author}(\texttt{p}_1, \texttt{a}_1): \quad AG(\texttt{AllPapersAssigned} \wedge \texttt{reviewer}(\texttt{p}_1, \texttt{a}_2) \rightarrow \neg K_{\texttt{a}_1}\texttt{reviewer}(\texttt{p}_1, \texttt{a}_2))$$

The proposition `AllPapersAssigned` in query 5.6 denotes a propositional formula that when evaluates to true, it shows all the papers are assigned to the reviewers. For a reader, it may look trivial that the property in query 5.6 does not hold in the system. Human brain uses reasoning to analyse the situation: reviewer $\texttt{a}_1$ has access to the number of papers assigned to reviewer $\texttt{a}_2$ and the list of assigned papers. So, if the papers of $\texttt{a}_2$ that show up for $\texttt{a}_1$ is less than the total number assigned to $\texttt{a}_2$, then $\texttt{a}_1$ knows that $\texttt{a}_2$ is the reviewer of $\texttt{p}_1$. In the above, the property holds when the knowledge modality K is treated as the knowledge gained by accessing the information. But when the knowledge is treated as reasoning about information, the property does not hold in the system.

Query 5.6 is not ACTLK as it contains the negation of knowledge modality. Therefore we apply an interactive refinement for the verification. In this approach, we build up the abstract model and refine it whenever a spurious strategy is found. Each time the property holds in the abstract model, the tool checks the local state of the agent $\texttt{a}_1$. If the local state is still abstract, we can not be sure that the property also holds in the concrete model. Hence, the tool returns the list of invisible local propositions for $\texttt{a}_1$. In each interactive refinement step, the user can select one or more propositions that he believes may help the agent in finding if $\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2)$ is true. Our interactive verification of query 5.6 proves that the property does not hold in the model. The final abstract model contained 28 Boolean variables compared to the 71 variable for the concrete one.

The concept of knowledge by reasoning can not be modelled by temporal properties. In the above query, $\mathtt{a}_1$ does not have direct access to the reviewer of his own paper. So, the verification frameworks like RW, PoliVer and DynPAL fail to show that the property in query 5.6 does not hold. Using interpreted systems make verifying such properties possible, which shows the value of the proposed approach.

The following query shows that reasoning about the assigned reviewer is always possible in our CRS:

Query 5.7.

$$\mathtt{author}(\mathtt{p}_1, \mathtt{a}_1) : \quad AG(\mathtt{AllPapersAssigned} \wedge \mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2) \rightarrow K_{\mathtt{a}_1}\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2))$$

Our tool proves that the property holds in the model in 10 refinement rounds. The initial abstraction begins with the proposition $\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2)$ and all the propositions included in the propositional formula of `AllPapersAssigned`. Each refinement step adds at least one proposition to the model. For 3 agents and 2 papers, the final abstract model contains 19 Boolean variables, while the number of variables in the concrete model is 71. This difference demonstrates the huge reduction of state space in the abstract model compared to the concrete one.

Some safety properties deal with detectability of an evidence in the system. As a practical example, in *EasyChair* conference paper review system, a PC member can update his profile information like email address. Updating user information is also possible by the chairs and can be done in a legal or illegal way. In all the cases, it is crucial that PC members can find out if their profile information in updated in order to trace illegal activities. This is a weakness in EasyChair implementation as the following safety property *does not* hold in easy chair:

Query 5.8.

$$\mathtt{PCMember}(\mathtt{a}_1) \wedge \neg\mathtt{profileUpdated}(\mathtt{a}_1) : AG(\mathtt{profileUpdated}(\mathtt{a}_1) \rightarrow K_{\mathtt{a}_1}\mathtt{profileUpdated}(\mathtt{a}_1))$$

This property is ACTLK and abstraction and refinement algorithm is applicable for its verification[1].

[1] We have not modelled easyChair in our implementation.

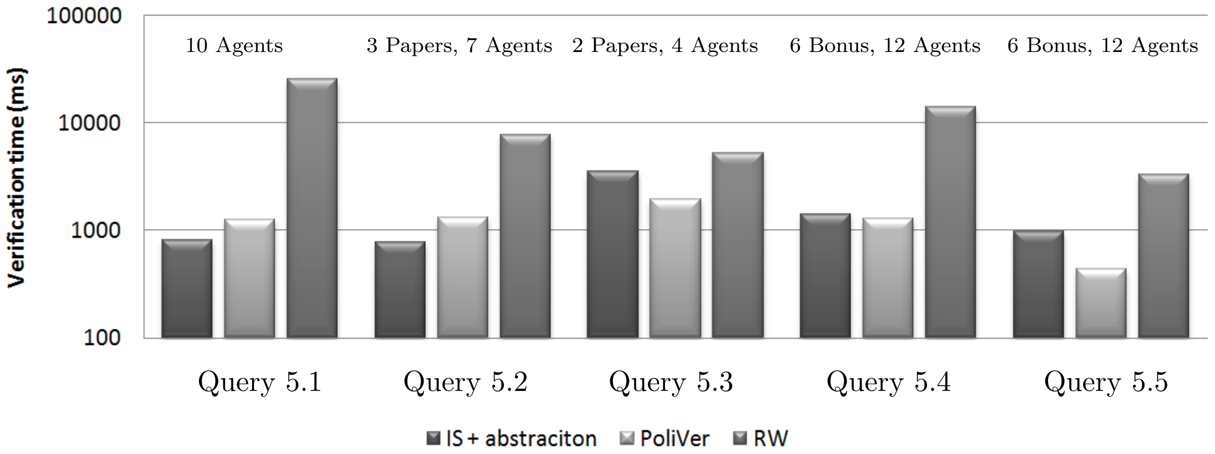

Figure 5.7: Comparison of the verification time for the queries 1 to 5 between MCMAS with abstraction, PoliVer and RW. Verification time for abstraction and refinement with MCMAS contains the time for model generation, invoking MCMAS (under Cygwin) and generating output. Queries 5.6 and 5.7 are not applicable for PoliVer and RW as they are not concerned about reasoning.

	Concrete model		Abstraction and refinement		
	time(s)	BDD vars	time(s)	Max BDD vars	last ref time
Query 5.6	6576.5	180	148.3	80	3.28
Query 5.7	6546.4	180	174.1	98	21

Figure 5.8: A comparison of query verification time (in second) and runtime memory usage (in MB) between the concrete model and automated abstraction refinement method. last ref time reflects the verification time for the last refined model.

5.6.5 Experimental results

One of the main motivations is to compare the memory usage and verification time in the concrete model and in the presence of abstraction. We performed the experiments on an Intel Core2 Dou 2.40GHz workstation with 2GB RAM running windows 7 64-bit.

Figure 5.7 demonstrates the comparison between interpreted systems model using abstraction and refinement, PoliVer and RW in a logarithmic scale. Except query 5.3, all the queries deal with temporal properties. Query 5.3 contains knowledge modality which we treat as knowledge by reading. Therefore, it is still possible to compare the tools for such a query. It is important to note that in abstraction and refinement method, a high percentage of evaluation time spends on generating the whole concrete model at the beginning, invoking executable MCMAS which also invokes Cygwin library, generating abstract model and verifying the counterexample. In most of our experiments, verification of the final abstract model by MCMAS takes less than 10ms.

Verification of the queries 5.6 and 5.7 by PoliVer and RW returns different results comparing with the interpreted systems model. PoliVer and RW are unable to detect information leakage in CRS policy because $\mathtt{author}(\mathtt{p}_1, \mathtt{a}_1)$ is always true in the model, and the agent $\mathtt{a}_1$ never finds a chance to sample $\mathtt{reviewer}(\mathtt{p}_1, \mathtt{a}_2)$. Therefore for the query 5.6, safety property holds in the system. Modelling in interpreted systems reveals that $\mathtt{a}_1$ can reason who is the reviewer of his paper. For query 5.6, the tool also outputs the counterexample which demonstrates the sequence of actions that allows the author to reason about the reviewer of his paper.

The unique feature of incorporating temporary read permissions into the interpreted systems and verifying with MCMAS has a big limitation: the number of evolution lines in MCMAS input script grows exponentially when knowledge for extra propositions is introduced. We have compared the memory usage and verification time for the queries 5.6 and 5.7 in the concrete model and the abstract one in figure 5.8. Comparing to the concrete model verification, the results show the considerable reduction in time and memory usage when applying the proposed abstraction and refinement method which shows its practical importance (fully automated refinement for the query 5.6 and interactive refinement for the query 5.7).

5.7 Summary

In this chapter, we introduced a framework for verifying temporal and epistemic properties over access control policies. In order to verify knowledge by reasoning, we used interpreted systems as the basic framework and MCMAS (model-checker for multi-agent systems) as the model-checking engine. Although we are able to find information leakage vulnerabilities in this approach, our experiments show that verifying the knowledge gained by reasoning increases the time and memory usage. In order to make the verification more practical for medium to large systems, we perform fully automated abstraction and refinement when dealing with safety properties. Our optimization method adopts counterexample-guided refinement known as CEGAR and extends it for ACTLK properties. Case studies and experimental results show a considerable reduction in time and space when abstraction and refinement is in use. We also apply an interactive refinement for some useful safety properties that does not reside in ACTLK like the ones that contain the negation of knowledge modality.

Most of the required properties that need to be verified are temporal or deal with the knowledge that is the result of reading system information. Those properties can be verified much more efficiently with PoliVer or DynPAL. This part of the research is the complimentary of our approach in chapter 4 i.e. we can find information leakage vulnerabilities that is difficult or impossible to be captured by PoliVer but uses more resources. We believe that both the approaches should be used together in order to prove that the policy complies with organization security requirements.

When defining the equivalence classes for the actions, we consider the agent that performs the action as a parameter for classification (definition 5.11). In general and when the model is too abstract, there may exists other agent's action resulting in a similar transition in the system which can be bundled in the same equivalence class and make the model simpler. Our approach makes the whole process of abstraction and refinement faster in practice, while it results in a bigger abstract model in model-checker's scripting language. But our experimental results show that the verification time of the abstract models construct a small portion of the whole verification process. Therefore, our approach results in faster process of abstract model generation, verification and refinement.

When the abstract model is small, there may exist several agents with similar behaviour in the system. As a possible enhancement, we can remove the redundant agents using symmetry reduction techniques. These techniques may also enable us in verifying models with unbounded number of agents. We leave this approach as future work. We would also like to work on a more intelligent heuristic when we find several candidates as the conflict clauses in future.

CHAPTER 6

INFORMATION LEAKAGE VERIFICATION IN DATALOG-BASED POLICIES

This chapter contains research on information leakage verification of access control policies which is done during my internship at Microsoft Research Cambridge as a part of my PhD. This research is done in collaboration with Moritz Becker, my supervisor in Microsoft Research. In this research, we were interested in verifying information leakage in stateless credential-based access control policies. We adopt Datalog as the policy language, which is the basis of various policy languages [70, 50, 51, 13]. We proposed the *first sound and complete algorithm* to find if a property is opaque (or detectable) in Datalog-based trust management system.

In 2009, Becker [12] introduced *probing attacks* in Datalog-based trust management systems, where an attacker submits a set of credentials together with access requests called *probes* and by analysing the response, reasons about confidential information. Becker proposed an inference system to verify the *detectability* of properties in a policy. There are some problems with the approach in [12]: (1) The detectability verification method is sound, but not provably complete (2) The method is found to be difficult to implement.

Considering the previous approach, we work on a verification method that fixes the problem of completeness and is feasible to automate. So, we design on a new approach which deals with opacity (also known as non-detectability). The inference system that we propose is not only sound (can detect if a property is opaque in the policy), but is also complete (if a property is opaque in a policy, the inference system will detect it). The algorithm is also simple enough to be automated. We have implemented the tool in F# functional language and introduced several optimization methods that effectively reduce the memory usage and calculation time of the algorithm.

This chapter is structured as follows: section 6.1 introduces the concept of probing attack in credential-based policies, section 6.2 provides the formal definitions, section 6.3 explains the structure of Datalog-based policies, in section 6.4, a delegation policy will be introduced, which will be the basis of out case studies, section 6.5 contains the opacity verification algorithm and corresponding lemmas and theorems, section 6.6 describes the implementation and optimization methods, section 6.7 is experimental results and summary is provided in section 6.8.

6.1 Introduction

To show how probing attacks work, consider a banking service which contains the following assertions in SecPAL policy syntax [13]:

Bank says x canAccessTransactions if x isClerk.
Bank says centralCA canSay x isClerk.

In the banking system, the central certificate authority is a trusted party, which can state (in practice by issuing a certificate) that a principal is a clerk. The keyword canSay in the second assertion states that bank has *delegated the authority* over the predicate isClerk to the centralCA.

The fact that an agent is an inspector in the banking system is confidential and not *visible* to the bank clerks. Now assume that Eve is interested in finding whether Bob is an inspector or not. Eve follows the following procedure:

1. By collaborating with centralCA, Eve owns two credentials:

   centralCA says Eve isClerk if Bob isInspector.

   centralCA says Bank canSay Bob isInspector.

2. For the first probe, Eve submits the two credentials together with the query "centralCA says Eve canAccessTransactions?", which is granted.

3. For the second probe, Eve submits the second credential together with the same query. The access is denied in this case.

Based on the above observations, Eve can find that the first credential is *crucial* for the successful access request. The credential affects the evaluation of the query only if the left hand side evaluates to true, which is only possible if Bank says Bob isInspector. Therefore by cleverly selecting the probes, an attacker may be able to reason about some confidential facts in the system.

In this research, we present a formal framework for probing attacks in credential systems. Similar to the work in this thesis for dynamic access control policies, we use the concept of *observational equivalence* to define opacity and detectability in access control policies. Given the set of available probes for an adversary, we present an algorithm to verify if a given query is opaque in a policy to the adversary. The algorithm is provably sound and complete.

Another important feature of the algorithm is that it provides the *witness* when the property is found to be opaque. Only the existence of a witness, which may be a non-logical policy is enough for a property to be opaque. But in the case of non-realistic witness, the attacker can ignore the witness or assign a *probability* which is an informal degree of likelihood, to the opacity of the property. The algorithm allows enumerating all possible witnesses for the opacity.

6.2 Probing attacks framework

Definition 6.1 (Policy, language, probe)**.** A *policy language* is a triple $(Pol, Prb, \vdash)$, where Pol and Prb are sets called *policies* and *probes*, respectively, and $\vdash$ is a binary infix relation from $Pol \times Prb$, called *decision relation.*

Let $A \in Pol$ and $\pi \in Prb$. If $A \vdash \pi$ we say that π is *positive in* A; otherwise $(A \nvdash \pi)$, π is *negative in* A.

This definition is abstract and does not enforce any restriction on the structure of policy and probe languages. A probe is a pair that contains a set of credentials and an access request, called query. A positive probe leads to an access grant, and a negative one leads to access denial.

Different policy languages like SecPAL [13], DKAL2 [51] and XACML [77] have different policy formats. For example, SecPAL policy is a set of assertions of the form $\langle Principal \rangle$ says $\langle Fact \rangle$ (see section 2.2.3 for more details). A probe π is of the form $\langle A, \varphi \rangle$ where A is the set of assertions and φ is the query. If A_0 is the system policy, then we say $A_0 \vdash \pi$ iff φ is deducible from $A_0 \cup A$.

The access queries in credential systems like DKAL2 and XACML are not easily evaluated against the union of system policy and submitted credentials. in DKAL2, the credentials called *infon terms* first will be modified into another form of credentials, and then will be added to the system policy. Also DKAL may filter out some of the credentials submitted by the user to the system according to the permissions. XACML policies have a hierarchical structure of *policy-sets*, *policies* and *rules* (refer to the section 2.2.1). Hence, the submitted credentials need to be transformed into the hierarchical structure before adding up to the policy.

To abstract away the language-dependent details like filtering assertions and translating the queries into other forms, we define the concept of available probes.

Definition 6.2 (Alikeness and available probes)**.** An *adversary* is defined by an equivalence relation $\simeq \subseteq Pol \times Pol$ and a set $Avail \subseteq Prb$ of available probes. If $A_1 \simeq A_2$ for two policies A_1 and A_2, we say that A_1 and A_2 are *alike.*

The definition of alikeness states that two policies are alike if the visible assertions are *syntactically* equivalent. There could be another definition of the alikeness which considers the *semantically equivalence* of the visible assertions.

Definition 6.3 (Observational equivalence)**.** Two policies A_1 and A_2 are *observationally equivalent* ($A_1 \equiv A_2$) iff

1. $A_1 \simeq A_2$, and

2. $\forall \pi \in Avail : A_1 \vdash \pi \iff A_2 \vdash \pi$

An attack to a policy can be of the type *passive* and *active.* A passive adversary only reads the visible assertions of system policy and can not distinguish between the policies that are alike. An active adversary is able to distinguish the policies not only by reading the visible assertions, but by evaluating his available probes against them. The two active and passive attacks over the policy provide a partial knowledge for the attacker, which can enable him to reason about confidential facts or rules in the system.

Definition 6.4 (Detectability, opacity)**.** A predicate $\Phi \subseteq Pol$ is *detectable* in $A \in Pol$ iff

$$\forall A' \in Pol : \; A \equiv A' \Rightarrow \Phi(A')$$

A predicate $\Phi \subseteq Pol$ is *opaque* in $A \in Pol$ iff it is not detectable in A, or in the other words, iff

$$\exists A' \in Pol : \; A \equiv A' \; \wedge \; \neg\Phi(A')$$

By the above definition, a property Φ is detectable is a policy A if Φ holds in all possible policies which behave the same against submitted probes. Otherwise, Φ is opaque in A. Opacity is the negation of detectability. A property is opaque in a policy if there exists another policy which is observationally equivalent to the first one, but Φ does not hold.

On the similarities between the knowledge-based verification of state-based policies and opacity verification in Datalog-based policies

The concept of alikeness and observational equivalence in credential-based systems is similar to the *epistemic accessibility relation* $\sim_i$ introduced in section 5.2.2. If s_1 and s_2 are two global states, then $s_1 \sim_i s_2$ if the observational part of s_1 and s_2 for agent i, which is called local state of i, is the same. In other words, agent i can not distinguish between the states s_1 and s_2.

The definition of detectability in trust management systems is closely related to the definition of knowledge which is denoted by operator K_i. Agent i knows if Φ in state s is true ($K_i\Phi$) if all the *reachable* states (similar to availability) that have the same local state as in s (observationally equivalent) satisfy the property Φ.

6.3 Datalog-based policies

Many existing policy languages including SecPAL [13], RT [70], Cassandra [17], SD3 [62] and Binder [41] use Datalog as their semantic bases. Datalog can be seen as Prolog without function symbols (see section 2.2.2 for more details).

The language is parameterized by a function-less first-order signature containing a countable set of predicate names and a countable set of constants. This gives rise to *atoms* P of the form $p(\vec{e})$, where p is a predicate symbol and $\vec{e}$ a sequence of *expressions* or *terms* (i.e., first-order variables or constants) of p's arity.

The central construct in Datalog is a *clause*. A clause a is of the form

$$P_0 \leftarrow P_1, \ldots, P_n$$

where $n \geq 0$. The atom P_0 is called the *head* and the sequence of atoms $\vec{P} = \langle P_1, \ldots, P_n \rangle$ the *body*. The arrow $\leftarrow$ is usually omitted if $n = 0$. We write *Cls* to denote the set of all clauses. We write $\mathbf{hd}(a)$ to denote a's head and $\mathbf{bd}(a)$ to denote its body. Given a set of clauses $A \subseteq \mathit{Cls}$, we write $\mathbf{hds}(A)$ to denote the atom set $\{\mathbf{hd}(a) \mid a \in A\}$.

A query φ is either **true**, **false** or a ground (i.e., variable-free) boolean formula (i.e., involving connectives $\neg$, $\wedge$ and $\vee$) over atoms. We write Qry to denote the set of all queries.

Given a query $\varphi \subseteq Qry$ and set of assertions A, we write $A \vdash \varphi$ if φ evaluates to true in A, and $A \not\vdash \varphi$ or equivalently $A \vdash \neg\varphi$ otherwise. In Datalog, $\vdash$ is the smallest relation such that the following holds:

- $A \vdash \mathbf{true}$.
- $A \vdash P_0$ if there exists atoms $P_1, \ldots, P_n$ (for some $n \geq 0$) such that $P_0 \leftarrow P_1, \ldots, P_n$ is a ground instance of some clause in A and $A \vdash P_i$ for all $i \in \{1, \ldots, n\}$.
- $A \vdash \neg\varphi$ if $A \vdash \varphi$ does not hold.
- $A \vdash \Phi \wedge \Phi'$ if $A \vdash \varphi$ and $A \vdash \varphi'$.

- $A \vdash \Phi \vee \Phi'$ if $A \vdash \varphi$ or $A \vdash \varphi'$.

For our proofs, a more operational definition is useful that is based on the intuition that Datalog clauses are inductive definitions.

Definition 6.5 (Consequence operator)**.** Given a policy A, we define the *consequence operator* T_A as a monotonic mapping between sets S of ground atoms. In the following definition, let γ be a ground substitution (a total mapping from variables to constants).

$$\begin{aligned} \mathsf{T}_A(S) = \{\ P_0' \ \mid\ & \exists\gamma\ \exists(P_0 < -P_1, ..., P_n) \in A, \\ & \gamma(\{P_1, ..., P_n\}) \subseteq S, \\ & P_0' = \gamma(P_0)\ \} \end{aligned}$$

The following definition introduces a number of terms that are fundamental to the algorithm described in Section 6.5. Lemma 6.1 establishes an important connection between probes and clause containment.

Definition 6.6 (Monotonicity, containment, equivalence)**.** A query is *monotonic* iff it is equivalent to one without negation. A probe $\langle A, \varphi \rangle \in \mathit{Prb}$ is *monotonic* iff φ is monotonic. A policy A is *contained in* a policy A' (we write $A \preceq A'$) iff for all ground atoms P and all sets S of ground atoms: $A \vdash \langle S, P \rangle \Rightarrow A' \vdash \langle S, P \rangle$. Two policies A and A' are *equivalent* (we write $A \doteq A'$) iff $A \preceq A'$ and $A' \preceq A$.

Lemma 6.1. Let $A \subseteq \mathit{Cls}$, $\vec{P}$ be a set of ground atoms and P a ground atom. $A \vdash \langle \vec{P}, P \rangle$ iff $\{P \leftarrow \vec{P}\} \preceq A$.

Proof.

($\Leftarrow$) This direction is straightforward.

($\Rightarrow$) Suppose the contrary. Let $n > 0$ be the smallest integer such that there exists a set S of ground atoms and a ground atom Q with $Q \in \mathsf{T}^n_{\{P \leftarrow \vec{P}\} \cup S}(\emptyset)$ and $Q \notin \mathsf{T}^w_{A \cup S}(\emptyset)$. Then there must be a ground instance $Q \leftarrow \vec{Q}$ of a clause in $\{P \leftarrow \vec{P}\} \cup S$ such that $\vec{Q} \subseteq \mathsf{T}^{n-1}_{\{P \leftarrow \vec{P}\} \cup S}(\emptyset) \subseteq \mathsf{T}^w_{A \cup S}(\emptyset)$. This clause cannot be in S, or else $Q \in \mathsf{T}^w_{A \cup S}(\emptyset)$.

Therefore, the ground instance is $P \leftarrow \vec{P}$, and thus $\vec{P} \subseteq \mathsf{T}^{n-1}_{\{P \leftarrow \vec{P}\} \cup S}(\emptyset) \subseteq \mathsf{T}^w_{A \cup S}(\emptyset)$. Hence $\mathsf{T}^w_{A \cup S}(\emptyset) = \mathsf{T}^w_{A \cup S \cup \vec{P}}(\emptyset)$. But from $A \vdash \langle \vec{P}, P \rangle$ it follows that $P \in \mathsf{T}^w_{A \cup S \cup \vec{P}}(\emptyset)$, and hence $P = Q \in \mathsf{T}^w_{A \cup S}(\emptyset)$, which contradicts the initial assumption. □

Now we can instantiate the abstract Definitions 6.1 and 6.2. For evaluating probes, we adopt the simple model where the query of a probe is evaluated against the union of the service's policy and the credentials (i.e., clauses) of the probe.

Definition 6.7 (Datalog instantiation)**.** We instantiate *Pol*to the powerset of clauses, $\wp(\mathit{Cls})$. A (Datalog) *policy* is hence a set $A_0 \subseteq \mathit{Cls}$. A (Datalog) *probe* π is a pair $\langle A, \varphi \rangle$, where $A \subseteq \mathit{Cls}$ and $\varphi \in \mathbf{Qry}$. Hence *Prb*is instantiated to the set of all such probes. A probe is *ground* iff it does not contain any variables. We write $\neg\langle A, \varphi \rangle$ to denote the probe $\langle A, \neg\varphi \rangle$. The *decision relation* $\vdash \subseteq \mathit{Pol} \times \mathit{Prb}$ is defined as follows:

$$A_0 \vdash \langle A, \varphi \rangle \Longleftrightarrow A_0 \cup A \vdash \varphi$$

Definition 6.8 (Adversary, Datalog alikeness)**.** An adversary is defined by a set $\mathit{Avail} \subseteq \mathit{Prb}$ and a unary predicate $\mathbf{Visible} \subseteq \mathit{Cls}$. If $\mathbf{Visible}(a)$ for some a $a \in \mathit{Cls}$, we say that a is *visible.*

We extend Visible to policies by defining the *visible part* of A, $\mathbf{Visible}(A)$, as $\{a \in A \mid \mathbf{Visible}(a)\}$, for all $A \subseteq \mathit{Cls}$.

Two policies $A_1, A_2 \subseteq \mathit{Cls}$ are *alike* ($A_1 \simeq A_2$) iff $\mathbf{Visible}(A_1) = \mathbf{Visible}(A_2)$.

Definition 6.9 (Probe detectability and opacity)**.** A probe $\pi \in \mathit{Prb}$ is *detectable* in $A \in \mathit{Pol}$ iff

$$\forall A' \in \mathit{Pol} : \; A \equiv A' \Rightarrow A' \vdash \pi$$

A probe $\pi \in \mathit{Prb}$ is *opaque in* $A \in \mathit{Pol}$ iff it is not detectable in A, or equivalently, iff

$$\exists A' \in \mathit{Pol} : \; A \equiv A' \; \wedge A' \nvdash \pi$$

This definition is a specialization of the definition 6.4 where Φ is instantiated with $\{A \subseteq \mathit{Cls} \mid A \vdash \pi\}$.

6.4 Example: a delegation policy

We define a realistic example of an authorization policy in Datalog, which also is used for our test cases later.

The example uses a grid computing scenario. The scenario consists of a compute cluster that allows users to run computing jobs. To execute a job, it may be required to access some data stored in a data centre. Both the policies of cluster and data centre, which govern who can run a job or access data *delegate authority* to a trusted third party.

The following is the set of assertions which represent the policy of the grid, where the first parameter of the predicate is the principal that "says" or in the other word, signs a fact. For instance, `canExec(Cluster,`x`,`j`)` means `Cluster` says principal x can execute the job j.

The first rule explains that principal x can execute job j on Cluster if x is member of cluster, x owns the job j (signed or accepted by Cluster), and data centre Data allows Cluster to read the job j:

```
canExec(Cluster,x,j) ←
   isMem(Cluster,x),
   owns(Cluster,x,j),                                    (1)
   canRead(Data,Cluster,j).
```

Cluster can delegate the authority over ownership and also membership to a trusted third party:

```
owns(Cluster,x,j) ←                                      (2)
   owns(y,x,j),isTTP(Cluster,y).
```

```
isMember(Cluster,x) ←                                    (3)
   isMember(y,x),isTTP(Cluster,y).
```

The data centre Data delegates the authority over the reading of data to the owner of the data:

```
canRead(Data,x,j) ←                                      (4)
   canRead(y,x,j),owns(Data,y,j).
```

Data can delegate the authority over ownership to a trusted third party:

```
owns(Data,x,j) ←                                         (5)
   owns(y,x,j),isTTP(Data,y).
```

In the scenario of grid policy, CA is known as a trusted third party. Therefore, the following facts also belong to the policy:

```
isTTP(Cluster,CA).                                      (6)
```

```
isTTP(Data,CA).                                         (7)
```

The policy A_0 consists of the assertions (1) - (7).

Now, assume that Eve possesses the following credentials issued by CA:

```
owns(CA,Eve,Job).                                       (8)
```

```
isMem(CA,Eve).                                          (9)
```

and Eve issued a credential (self-issued) that allows Cluster to read Job:

```
canRead(Eve,Cluster,Job).                              (10)
```

Now, the mission of Eve is to find out if Bob is a member of the Cluster. Hence, she issues another self-issued credential to which allows Cluster to read the job Job with the condition that Bob is a member of Cluster:

$$\texttt{canRead(Eve,Cluster,Job)} \leftarrow \qquad (11)$$
$$\quad \texttt{isMem(Cluster,Bob).}$$

The set of credentials that is possessed by Eve denoted by $A_{\texttt{Eve}}$ is the credentials (8) - (11). The set of available probes that Eve can run against A_0 is

$$Avail = \{\langle A, \varphi_{\texttt{Eve}}\rangle \mid A \subseteq A_{\texttt{Eve}}\}$$

where $\varphi_{\texttt{Eve}} = \texttt{canExec(Cluster,Eve,Job)}$.
We assume that no assertion in the policy is visible for Eve, or equivalently *visible* $= \emptyset$.

The observations of Eve are as follows:

1. The probe $\langle A_{\texttt{Eve}}, \varphi_{\texttt{Eve}}\rangle$ is positive in A_0. In the other words $A_0 \vdash \langle A_{\texttt{Eve}}, \varphi_{\texttt{Eve}}\rangle$. The derivation goes in the following way:

   (a) isMem(Cluster,Eve) which denotes the membership of Eve in Cluster is deducible from (6), (9) and (3).

   (b) owns(Cluster,Eve,Job) is deducible from (6), (8) and (2).

   (c) owns(Data,Eve,Job) is implied from (8), (7) and (5).

   (d) canRead(Data,Cluster,Job) is deducible from (10), (c) and (4).

   (e) The query canExec(Cluster,Eve,Job) implies from (a), (b) and (d).

2. $A_0 \vdash \langle\{(8) - (10)\}, \varphi_{\texttt{Eve}}\rangle$. In the context of this thesis, a probe similar to this one is called *minimally positive*, where any strictly smaller set of assertions result in a negative probe.

3. $A_0 \nvdash \langle A, \varphi_{\texttt{Eve}}\rangle$ for all $A \subseteq \{(8), (9), (11)\}$. In the other words, replacing probe (10) in item 2) with (11) results in a negative probe.

4. All the policies A_0' that are observationally equivalent to A_0 satisfy the property $A_0' \nvdash$ isMem(Cluster,Bob). Assume the contrary where $A_0' \vdash$ isMem(Cluster,Bob). By item 2, we know that $\varphi_{\texttt{Eve}}$ holds in $A_0' \cup \{(8) - (10)\}$. If we replace the clause (10) with (11), $\varphi_{\texttt{Eve}}$ still holds in $A_0' \cup \{(8), (9), (11)\}$ as the body of the clause (11) is true in A_0' and therefore, the replacement does not make any difference. But this contradicts item 3).

5. It is clear that the probe $\langle \emptyset, \neg\texttt{isMem(Cluster,Bob)}\rangle$ is detectable in A_0. Note that this probe is not available for `Eve`.

6. The probe $\pi = \langle \{(9),(11)\}, \texttt{canRead(Data,Cluster,Job)}\rangle$ is opaque in A_0. The opacity of π is not trivial as $A_0 \vdash \pi$. But there exists a policy A'_0 build by removing clause (4) and replacing `Data` in clause (1) by x, which $A'_0 \nvdash \pi$.

6.5 Opacity verification algorithm

Given a DATALOG policy $A_0 \subseteq \mathit{Cls}$ and an adversary which is defined by a set of available probes $\mathit{Avail} \subseteq \mathit{Prb}$ and visibility function **Visible** where $\textbf{Visible}(A_0) \subseteq A_0$, the algorithm determines if a given ground probe $\pi_0 \in \mathit{Prb}$ is opaque (or detectable) in A_0. The limitation of the algorithm is the ground input probes, which is reasonable in real applications as the probes are generally issued for a specific principal and purpose.

As discussed before, a probe π is opaque in A_0 iff there exists a policy A'_0 which is observationally equivalent to A_0 and π is negative in A'_0. Therefore, to prove the opacity of a probe, we attempt to construct A'_0 as the witness. To prove if π is detectable, we show that no such A'_0 exists.

6.5.1 Query decomposition

Consider a probe $\pi = \langle A, \varphi_1 \vee \varphi_2\rangle \in \mathit{Avail}$ which is positive in A_0. We look for the policy A'_0 such that $A'_0 \vdash \pi$ or equivalently $A'_0 \cup A \vdash \varphi_1 \vee \varphi_2$. Therefore, it is equivalent to finding A'_0 such that $A'_0 \vdash \langle A, \varphi_1\rangle$ or the one in which $A'_0 \vdash \langle A, \varphi_2\rangle$. So, a disjunction in the query of a probe results is a branch in the search of A'_0. In the case of negative probes in A_0, since $A_0 \nvdash \pi$ is equivalent to $A_0 \vdash \neg\pi$, we convert all the negative probes to positive ones and then deal with the disjunctions.

The algorithm starts by computing all disjunctive branches by first computing the equivalent disjunctive normal form of the queries in the set of available probes. Then the algorithm constructs a Cartesian product of the disjuncts.

Definition 6.10 (Disjunctive normal form)**.** Let $\mathbf{dnf}(\varphi)$ denote the disjunctive normal form of the query φ, encoded as a set of pairs (S^+, S^-) of sets of atoms. So

$$\varphi \iff \bigvee_{(S^+,S^-)\in\mathbf{dnf}(\varphi)} (\bigwedge S^+ \wedge \neg \bigvee S^-)$$

Example 6.1. Let $\varphi = (p\wedge q\wedge\neg s)\vee(\neg p\wedge\neg q\wedge s)$. Then $\mathbf{dnf}(\varphi) = \{(\{p,p\}, s), (\{s\}, \{p,q\})\}$.

Lemma 6.2. Let $\langle A, \varphi\rangle \in \mathit{Prb}$. Then $A_0 \vdash \langle A, \varphi\rangle$ iff

$$\exists(S^+, S^-) \in \mathbf{dnf}(\varphi) : A_0 \vdash \langle A, \bigwedge S^+\rangle \text{ and } A_0 \not\vdash \langle A, \bigvee S^-\rangle$$

The function $\mathbf{flatten}_{A_0}$ accepts a set of probes. For each probe in the set, if the probe is negative in A_0, it converts it to a positive probe by negating the query inside the probe. $\mathbf{flatten}_{A_0}$ constructs a set of pairs of probe sets, where each pair corresponds to a set of disjunctive search branch.

Definition 6.11 (Flatten)**.** Let $\Pi \subseteq \mathit{Prb}$. Then $\mathbf{flatten}_{A_0}(\Pi)$ is a set of pairs (Π^+, Π^-) of sets of probes defined inductively as follows:

$$\begin{aligned}
&\mathbf{flatten}_{A_0}(\emptyset) = \{(\emptyset, \emptyset)\}.\\
&\mathbf{flatten}_{A_0}(\Pi \cup \{\langle A, \varphi\rangle\}) = \{(\Pi^+, \Pi^-) \mid\\
&\quad \exists(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\Pi), (S^+, S^-) \in \mathbf{dnf}(\widetilde{\varphi}) :\\
&\qquad \Pi^+ = \Pi_0^+ \cup \{\langle A, \bigwedge S^+\rangle\} \text{ and}\\
&\qquad \Pi^- = \Pi_0^- \cup \{\langle A, \bigvee S^-\rangle\}\}\\
&\quad \text{where } \widetilde{\varphi} = \varphi \text{ if } A_0 \vdash \langle A, \varphi\rangle, \text{ and } \widetilde{\varphi} = \neg\varphi \text{ otherwise.}
\end{aligned}$$

Lemma 6.3. Let $A_0' \subset \mathit{Cls}$ and $\Pi \subseteq \mathit{Prb}$.

$$\begin{aligned}
&\forall \pi \in \Pi : A_0' \vdash \pi \iff A_0 \vdash \pi \text{ iff}\\
&\quad \exists(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\Pi) : \quad (\forall \pi \in \Pi_0^+ : A_0' \vdash \pi) \text{ and } (\forall \pi \in \Pi_0^- : A_0' \not\vdash \pi)
\end{aligned}$$

Proof. ($\Rightarrow$) Assume $\forall \pi \in \Pi : A_0' \vdash \pi \iff A_0 \vdash \pi$. For the sake of contradiction, suppose the negation of the right hand side where true. By the definition of flatten, there exits $\pi = \langle A_\pi, \varphi\rangle$ and $(S^+, S^-) \in \mathbf{dnf}(\widetilde{\varphi})$ (where $\widetilde{\varphi} = \varphi$ if $A_0 \vdash \langle A, \varphi\rangle$, and $\widetilde{\varphi} = \neg\varphi$ otherwise) such that $A_0' \not\vdash \langle A_\pi, \bigwedge S^+\rangle$ or $A_0' \vdash \langle A_\pi, \bigvee S^-\rangle$.

If $A_0 \models \pi$, then by Lemma 6.2 we have $A_0' \not\models \pi$ which contradicts the initial assumption. If $A_0 \not\models \pi$ then by Lemma 6.2 we have $A_0' \not\vdash \langle A_\pi, \neg\varphi\rangle$ and hence $A_0' \vdash \pi$, which is again a contradiction. Therefore, the right hand side of the equivalence holds.

($\Leftarrow$) Assume that the right hand side of the equivalence is correct. Let $\pi = \langle A_\pi, \varphi\rangle \in \Pi$. Suppose $A_0 \vdash \pi$. Then by the assumption and the definition of **flatten**, there exists $(S^+, S^-) \in \mathbf{dnf}(\varphi)$ such that $A_0' \vdash \langle A_\pi, \bigwedge S^+\rangle$ and $A_0' \not\vdash \langle A_\pi, \bigvee S^-\rangle$. From Lemma 6.2 it follows that $A_0' \vdash \pi$ as required. Now suppose $A_0 \not\vdash \pi$. Then we can show that $A_0' \vdash \langle A_\pi, \neg\varphi\rangle$ and hence $A_0' \not\vdash \pi$ as required.

□

Example 6.2. Suppose $\mathit{Avail} = \{\pi_1, \pi_2\}$ where $\pi_1 = \langle A_1, \neg p \vee q\rangle$ and $\pi_2 = \langle A_2, \neg p \wedge q\rangle$.

Suppose further that $A_0 \vdash \pi_1$ and $A_0 \not\vdash \pi_2$. Let $\pi_1^p = \langle A_1, p\rangle$, $\pi_1^q = \langle A_1, q\rangle$, $\pi_2^p = \langle A_2, p\rangle$ and $\pi_2^q = \langle A_2, q\rangle$. Then $\mathbf{flatten}_{A_0}(\mathit{Avail})$ contains four pairs of the probe sets: $(\{\pi_2^p\}, \{\pi_1^p\})$, $(\emptyset, \{\pi_1^p, \pi_2^q\})$, $(\{\pi_1^q, \pi_2^p\}, \emptyset)$, $(\{\pi_1^q\}, \{\pi_2^q\})$

Apart from observational equivalence, opacity also requires that $\pi_0 = \langle A, \varphi\rangle$ be negative in A_0'. By Lemma 6.2, this is equivalent to picking a pair $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$ such that $A_0' \vdash \langle A, \bigwedge S^+\rangle$ and $A_0' \not\vdash \langle A, \bigvee S^-\rangle$.

Let $\pi_0 = \langle A, \varphi\rangle$ be a probe and $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$. The probe π_0 is opaque in the policy A_0 if there exists A_0' such that:

1. $A_0' \simeq A_0$
2. There exists a pair $(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\mathit{Avail})$ such that:
   (a) all probes in $\Pi_0^+ \cup \{\langle A, \bigwedge S^+\rangle\}$ are positive, and
   (b) all probes in $\Pi_0^- \cup \{\langle A, \bigvee S^-\rangle\}$ are negative.

If such A_0' exists, then we call A_0' as *witness* for the opacity of π_0 in A_0.

The high level overview of the search strategy is as follows: taking the requirement of policy alikeness, the algorithm starts with the policy $\mathbf{Visible}(A_0)$. To satisfy the requirement (a), we go through each probe in Π^+ one by one and for each probe and we add one or more clauses to the witness candidate. Monotonicity of $\vdash$ guarantees that addition of the clauses does not violate (a). To satisfy the requirement (b), after each addition, we check if (b) still holds. If not, we need to backtrack and try a different way to satisfy (a). When all probes of Π^+ have been considered, the candidate is guaranteed to be a witness.

6.5.2 Preserving alikeness

Addition of clauses may violate the alikeness requirement $A_0' \simeq A_0$ when the newly added clause $a \notin A_0$ while $\mathbf{Visible}(a) = \top$. To ensure that only invisible clauses are added, we use a nullary predicate p_{Hi} that does not occur in A_0, nor in π_0 nor in Avail. Therefore, if p_{Hi} occurs in clause $a \in \mathit{Cls}$, then $\neg\mathbf{Visible}(a)$. Addition of p_{Hi} as a freshly chosen predicate to the witness does not make any difference in terms of observationally equivalence. Therefore, instead of adding $P \leftarrow \vec{P}$ to the witness, we add $P \leftarrow p_{\mathrm{Hi}}, \vec{P}$ which preserves the alikeness. The following Lemma formalizes the discussion:

Lemma 6.4. Let $A \subseteq \mathit{Cls}$ such that p_{Hi} does not occur in A. Then there exists $\hat{A} \subseteq \mathit{Cls}$ such that $\hat{A} \doteq A \cup \{p_{Hi}\}$ and $\mathbf{Visible}(\hat{A}) = \emptyset$.

$$
(\textsc{Init})\ \frac{\begin{array}{c}(\Pi_0^+,\Pi_0^-)\in\mathbf{flatten}_{A_0}(\Pi_0)\qquad \pi_0=\langle A,\varphi\rangle\\ (S^+,S^-)\in\mathbf{dnf}(\neg\varphi)\qquad \forall\pi\in\Pi^-\cup\{\langle A,\bigvee S^-\rangle\}:\mathbf{Visible}(A_0)\not\vdash\pi\end{array}}{\langle\Pi^+\cup\{\langle A,\bigwedge S^+\rangle\},\Pi^-\cup\{\langle A,\bigvee S^-\rangle\},\mathbf{Visible}(A_0)\rangle\in\mathbf{Init}(\Pi_0)}
$$

$$
(\textsc{Probe})\ \frac{\begin{array}{c}\widetilde{A}\subseteq A\\ \langle a_1,\ldots,a_n\rangle\in\mathbf{perms}(\widetilde{A})\qquad\forall i\in\{1,\ldots,n\}:\vec{P}_i=\mathbf{bd}(a_i)\qquad\vec{P}_{n+1}=\vec{P}\\ A''=\bigcup_{k=1}^{n+1}\bigcup_{P_k\in\vec{P}_k}\{P_k\leftarrow\mathbf{hds}(\{a_1,\ldots,a_{k-1}\})\}\qquad\forall\pi\in\Pi^-:A'\cup A''\not\vdash\pi\end{array}}{\langle\Pi^+\cup\{\langle A,\bigwedge\vec{P}\rangle\},\Pi^-,A'\rangle\xrightarrow{\langle A,\bigwedge\vec{P}\rangle}\langle\Pi^+,\Pi^-,A'\cup A''\rangle}
$$

Figure 6.1: Transition system for verifying opacity

Proof. Let $\hat{A}=\{(P\leftarrow p_{Hi},\vec{P})\mid(P\leftarrow\vec{P})\in A\}\cup\{p_{Hi}\}$. Consider any atom P and set of atoms $\vec{P}$, and let $A_1=\{P\leftarrow p_{Hi},\vec{P}\}\cup\{p_{Hi}\}$ and $A_2=\{P\leftarrow\vec{P}\}\cup\{p_{Hi}\}$. A simple induction on n shows that for all sets S of ground atoms, $\mathbf{T}^n_{A_1\cup S}(\emptyset)=\mathbf{T}^n_{A_2\cup S}(\emptyset)$, and hence, $A_1\doteq A_2$. Therefore, $\hat{A}\doteq A\cup\{p_{Hi}\}$.

□

6.5.3 Initial states

The algorithm is presented in a non-deterministic state transition system where a state is a triple $\langle\Pi^+,\Pi^-,A\rangle$. Π^+ and Π^- are sets of ground probes and A is the policy. Π^+ is the set of positive probes that are not considered yet, Π^- are the set of probes that should be negative in the witness and A is the witness candidate. Each state transition (non-deterministically) removes a positive probe from Π^+ and adds the resulting clauses to the witness. A *final state* is of the form $\langle\emptyset,\Pi^-,A_0'\rangle$. If the final state is produced by a series of transitions starting from an initial state, the policy A_0' is guaranteed to be a witness for the opacity of π_0 in A_0, and such a final state exists iff π_0 is opaque in A_0.

Definition 6.12 (Initial state)**.** The rule (Init) in Fig. 6.1 defines a set $\mathbf{Init}(\Pi_0)$ of states, parametrised by a set of probes Π_0. We write **Init** to denote $\mathbf{Init}(\mathit{Avail})$, the set of *initial states.*

Lemma 6.5 (Soundness and completeness of (Init))**.** The following two statements are equivalent:

1. π_0 is opaque in A_0.

2. There exists $\langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0)\rangle \in \mathbf{Init}$ and $A_0' \subseteq \mathit{Cls}$ such that p_{Hi} does not occur in A_0' and $\mathbf{Visible}(A_0) \preceq A_0'$ and

$$\forall \pi \in \Pi^+ : A_0' \vdash \pi \text{ and } \forall \pi \in \Pi^- : A_0' \nvdash \pi$$

Proof. ($1 \Rightarrow 2$) If $A_0 \nvdash \pi_0$ then the statement holds for $A_0' = A_0$. Now assume that $A_0 \vdash \pi_0$ and let $\pi_0 = \langle A_\pi, \varphi_\pi \rangle$. Then there exists $A_0' \in \mathit{Cls}$ such that p_{Hi} does not occur in A_0' and $\mathbf{Visible}(A_0) \preceq A_0'$ and

$\forall \pi \in \mathit{Avail} : A_0' \vdash \pi \iff A_0 \vdash \pi$ and

$A_0' \vdash \neg \pi_0$

By the definition of opacity and Lemma 6.3:

$$\exists (\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\mathit{Avail}) : \quad (\forall \pi \in \Pi_0^+ : A_0' \vdash \pi) \text{ and } (\forall \pi \in \Pi_0^- : A_0' \nvdash \pi)$$

By Lemma 6.2, there exists $(S^+, S^-) \in \mathbf{dnf}(\neg \varphi_\pi)$ such that $A_0' \vdash \{\langle A_\pi, \bigwedge S^+ \rangle\}$ and $A_0' \nvdash \{\langle A_\pi, \bigvee S^- \rangle\}$. By (Init), $\langle \Pi^+ \cup \{\langle A_\pi, \bigwedge S^+\rangle\}, \Pi^- \cup \{\langle A_\pi, \bigvee S^- \rangle\}, \mathbf{Visible}(A_0)\rangle \in \mathbf{Init}$ as required by 2).

($2 \Rightarrow 1$) If $A_0 \nvdash \pi_0$, then 1) is trivially true. Now assume $A_0 \vdash \pi_0$. Then by (Init) and Lemmas 6.3 and 6.2:

$\forall \pi \in \mathit{Avail} : A_0' \vdash \pi \iff A_0 \vdash \pi$ and

$A_0' \vdash \neg \pi_0$

Since p_{Hi} occurs neither in *Avail* nor in $\neg \pi_0$, there exists $\hat{A}_0 \subseteq \mathit{Cls}$ such that, by Lemma 6.4, $\hat{A}_0 \doteq A_0' \cup p_{Hi}$ and $\mathbf{Visible}(\hat{A}_0) = \emptyset$.

Now consider $\hat{A}_0' = \hat{A}_0 \cup \mathbf{Visible}(A_0)$. Clearly, $\hat{A}_0' \simeq A_0$. By (Init), $A = \mathbf{Visible}(\hat{A}_0)$, so by assumption, $\mathbf{Visible}(A_0) \preceq A_0' \preceq \hat{A}_0$. Therefore, $\hat{A}_0' \doteq \hat{A}_0$.

Since p_{Hi} occurs neither in A_0' nor in π_0 nor in *Avail*l, we have

$\forall \pi \in \mathit{Avail} : \hat{A}_0' \vdash \pi \iff A_0 \vdash \pi$ and

$\hat{A}_0' \vdash \neg \pi_0$

Hence $\hat{A}_0' \equiv A_0$ and $\hat{A}_0' \nvdash \pi_0$, as required. □

6.5.4 Algorithm termination: minimal witnesses

Potentially, for every probe $\langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0)\rangle \in \mathbf{Init}$ there are infinitely many $A_0' \succeq \mathbf{Visible}(A_0)$ such that all the probes in Π^+ are positive. To prove the termination of algorithm, we need to get sure there is finite number of searches for proving that π_0 is *not* opaque.

Our algorithm computes a finite set of *minimal* witness candidates that is sufficient to prove the opacity. The candidates are minimal in the sense that for every policy A'_0 that makes the probes in Π^+ positive, there exists a policy A''_0 in the set of computed witnesses in which $A''_0 \preceq A'_0$. By the anti-monotonicity of $\not\vdash$, if A'_0 makes the probes in Π^- negative, A''_0 also makes them negative. This property is the basis for the prove of completeness and termination of the algorithm.

6.5.5 Computing the witnesses

The transition system considers one positive probe in each state transition. Consider a state $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A' \rangle$. We aim to find all minimal ways of extending A' to some $A' \cup A''$ such that $A' \cup A'' \vdash \pi$. By the monotonic construction of positive probes, we can ignore A' and simply find all minimal A'' such that $A'' \vdash \pi$.

Assume that $\pi = \langle A, \varphi \rangle$ is positive in A''. Then there should exists a subset of clauses $\widetilde{A} \subseteq A$ which are relevant in making the probe positive. To build up the witness, we need to consider all $2^{|A|}$ possible cases, since each $\widetilde{A}$ results in a different witness that are incomparable by the relation $\preceq$. The choice of the subset in each state results in a different state. Therefore, the transition system is non-deterministic.

Now assume the positive probe $\pi = \langle A, \bigwedge \vec{P} \rangle \in \Pi^+$. We look for the minimal clauses A'' under the assumption that $\widetilde{A} \subseteq A$ is relevant. Since $\widetilde{A}$ is relevant, all the clauses $P_0 \leftarrow P_1, \ldots, P_n \in \widetilde{A}$ are actively involved in the derivation of $A'' \cup \widetilde{A} \vdash \bigwedge \vec{P}$. This is possible if the body atoms are derivable, and derivation of $\bigwedge \vec{P}$ depends on all the heads of clauses in $\widetilde{A}$. Lets demonstrate the process with an example:

Example 6.3. Suppose that $\vec{P} = z$ and $\widetilde{A} = \{p \leftarrow q., r \leftarrow s., u \leftarrow v.\}$. We aim to find all minimal A'' such that $A'' \cup \widetilde{A} \vdash z$.

The simplest case is when all the body atoms in $\widetilde{A}$ is true in A'' and z is derived from the heads in $\widetilde{A}$ (stage 1):

$A''_1 = \{q., s., v., z \leftarrow p, r, u\}$

For the next stage, we assume all the configurations in which A'' contains the body atoms of $\widetilde{A}$' clauses, and the heads of the clauses combine with the clauses in A'' to make the body atoms in $\widetilde{A}$ true.

We have six other solutions, where in three of them, A'' contains only one body atom, and in other three, A'' contains two body atoms. Two of the solutions are as follows:

$A''_2 = \{q., s \leftarrow p., v \leftarrow p., z \leftarrow p, r, u\}$
$A''_5 = \{q., s., v \leftarrow p, r., z \leftarrow p, r, u\}$

For our example, we have the following six solutions for different permutations of $\widetilde{A}$, in which *are contained* in all other candidates for A'':

$A''_8 = \{q., s \leftarrow p., v \leftarrow p, r., z \leftarrow p, r, u\}$

$A''_9 = \{q., v \leftarrow p., s \leftarrow p, u., z \leftarrow p, r, u\}$

$A''_{10} = \{s., q \leftarrow r., v \leftarrow p, r., z \leftarrow p, r, u\}$

$A''_{11} = \{s., v \leftarrow r., q \leftarrow r, u., z \leftarrow p, r, u\}$

$A''_{12} = \{v., q \leftarrow u., s \leftarrow p, u., z \leftarrow p, r, u\}$

$A''_{13} = \{v., s \leftarrow u., q \leftarrow r, u., z \leftarrow p, r, u\}$

The solution A''_8 is contained in ($\preceq$) A''_1, A''_2 and A''_5. For each solution in $\{A''_1, \ldots, A''_7\}$, there exists a solution from $\{A''_8, \ldots, A''_{13}\}$ which is contained in the first one. So, we only require considering the solutions in stage three. This is because if one of the clause in the other solutions makes all the probes in Π^- negative, then this will be the case for the corresponding solution in stage three. It is possible to show that this observation holds in the general case [14].

Lemma 6.6 (Soundness of $\xrightarrow{\pi}$)**.** If $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A' \rangle \xrightarrow{\pi} \langle \Pi^+, \Pi^-, A' \cup A'' \rangle$, then $A'' \vdash \pi$.

Proof. By (Probe), π must be of the form $\langle A, \bigwedge \vec{P} \rangle$. We will prove the following, stronger, statement.

For all $\widetilde{A} \subseteq A$, $\langle a_1, \ldots, a_n \rangle \in \mathsf{Perm}(\widetilde{A})$ and

$$A''_0 = \bigcup_{k=1}^{n} \bigcup_{P_k \in \mathsf{bd}(a_k)} \{P_k \leftarrow \mathsf{hds}(\{a_1, ..., a_{k-1}\})\} \tag{6.1}$$

we have $A''_0 \cup \widetilde{A} \vdash \bigwedge \mathsf{hds}(\widetilde{A})$.

This implies the statement of the lemma since, by the definition of (Probe), $A'' = A''_0 \cup \{P \leftarrow \mathsf{hds}(\widetilde{A})\}$ is precisely the set of assertions added to the state in a $\xrightarrow{\pi}$ transition; furthermore, for all $P \in \vec{P}$: $(P \leftarrow \mathsf{hds}(\widetilde{A})) \in A''$ and $\widetilde{A} \subseteq A$, and therefore:

$$A'' \cup \widetilde{A} \vdash \bigwedge \mathsf{hds}(\widetilde{A}) \Rightarrow A'' \cup A \vdash \bigwedge \vec{P} \Rightarrow A'' \vdash \pi. \tag{6.2}$$

The proof proceeds by induction on n. In the base case, $\widetilde{A} = \mathsf{hds}(\widetilde{A}) = \emptyset$. Then $\mathsf{hds}(\widetilde{A}) = \emptyset$, so the statement trivially holds.

In the inductive case, $n = |\widetilde{A}| > 0$. Consider any $\langle a_1, \ldots, a_n \rangle \in \mathsf{Perm}(\widetilde{A})$. Let A''_0 be defined as in (6.1). Then we have $A''_0 = A''_1 \cup A''_2$, where

$$A''_1 = \bigcup_{k=1}^{n-1} \bigcup_{P_k \in \mathsf{bd}(a_k)} \{P_k \leftarrow \mathsf{hds}(\{a_1, ..., a_{k-1}\})\},$$
$$A''_2 = \bigcup_{P_k \in \mathsf{bd}(a_n)} \{P_k \leftarrow \mathsf{hds}(\{a_1, ..., a_{n-1}\}\})\}.$$

By the induction hypothesis, $A''_1 \cup \{a_1, \ldots, a_{n-1}\} \vdash \bigwedge \mathsf{hds}(\{a_1, \ldots, a_{n-1}\})$. By monotonicity of $\vdash$, it also holds that

$$A''_0 \cup \widetilde{A} \vdash \bigwedge \mathsf{hds}(\{a_1, \ldots, a_{n-1}\}). \tag{6.3}$$

It remains to show that $A''_0 \cup \widetilde{A} \vdash \mathsf{hd}(a_n)$.

By the definition of A''_2 and by (6.3), and since $A''_2 \subseteq A''_0$, we have $A''_0 \cup \widetilde{A} \vdash \bigwedge \mathsf{bd}(a_n)$. Since $a_n = (\mathsf{hd}(a_n) \leftarrow \mathsf{bd}(a_n)) \in \widetilde{A}$, this implies $A''_0 \cup \widetilde{A}' \vdash \mathsf{hd}(a_n)$, as required. □

Lemma 6.7 (Completeness of $\xrightarrow{\pi}$)**.** Let A_1 and A_2 be the policies with $n = |A_2|$, and $\vec{P}$ a set of ground atoms. If $A_1 \vdash \langle A_2, \bigwedge \vec{P} \rangle$ and for all $A'_2 \subsetneq A_2 : A_1 \not\vdash \langle A'_2, \bigwedge \vec{P} \rangle$, then there exists $\langle a_1, \ldots, a_n \rangle \in \mathbf{perms}(A_2)$ and

- $\vec{P}_i = \mathbf{bd}(a_i)$, for $i \in \{1, \ldots, n\}$, and
- $\vec{P}_{n+1} = \vec{P}$

such that for all $i \in \{1, \ldots, n+1\}$:

$$\forall P_i \in \vec{P}_i : (P_i \leftarrow \mathbf{hds}(\{a_1, \ldots, a_{i-1}\})) \preceq A_1$$

Proof. Let $\Phi(A_1, A_2, \vec{P})$ denote the parametrized statement of the lemma. The proof proceeds by induction on $n = |A_2|$. If $n = 0$, the statement trivially holds.

Now assume $n > 0$. By assumption, A_2 is minimal. Therefore, there exists a smallest integer m such that $\mathsf{hds}(A_2) \subseteq \mathsf{T}^m_{A_1 \cup A_2}(\emptyset)$. Furthermore, $m \geq 1$, since $n > 0$. Hence there exists $\widetilde{A}_2$, the largest subset of A_2 such that $\mathsf{hds}(\widetilde{A}_2) \cap \mathsf{T}^{m-1}_{A_1 \cup A_2}(\emptyset) = \emptyset$, and such that $k = |\widetilde{A}_2| \geq 1$.

Let $\vec{P}'$ be the set of all body atoms of clauses in $\widetilde{A}_2$. By construction of $\widetilde{A}_2$ and since $\mathsf{hds}(\widetilde{A}_2) \subseteq \mathsf{hds}(A_2) \subseteq \mathsf{T}^m_{A_1 \cup A_2}(\emptyset)$, we have that $\vec{P}' \subseteq \mathsf{T}^{m-1}_{A_1 \cup A_2}(\emptyset)$, and again by construction of $\widetilde{A}_2$, we also have $\vec{P}' \subseteq \mathsf{T}^{m-1}_{A_1 \cup (A_2 \setminus \widetilde{A}_2)}(\emptyset)$. Since $|A_2 \setminus \widetilde{A}_2| = n - k < n$, we can assume the inductive hypothesis $\Phi(A_1, A_2 \setminus \widetilde{A}_2, \vec{P}')$, and in particular, the existence of $\langle a_1, ..., a_{n-k} \rangle \in \mathsf{Perm}(A_2 \setminus \widetilde{A}_2)$ with the stated properties.

Let $\langle a_{n-k+1}, ..., a_n \rangle$ be any permutation of $\widetilde{A}_2$. We thus have constructed a permutation $\langle a_1, ..., a_n \rangle \in \mathsf{Perm}(A_2)$. This permutation gives rise to sets $S_0, ..., S_n$ and $\vec{P}_1, ..., \vec{P}_{n+1}$ as defined in the lemma.

For $i \in \{1, ..., n-k\}$, the desired property follows directly from the inductive hypothesis. Furthermore, the inductive hypothesis states that

$$\forall P' \in \vec{P'} : (P' \leftarrow \mathsf{hds}(\{a_1, ..., a_{n-k}\})) \preceq A_1.$$

Then for $i \in \{n-k+1, ..., n\}$, we get (since $\vec{P_i} \subseteq \vec{P'}$):

$$\forall P_i \in \vec{P_i} : (P_i \leftarrow \mathsf{hds}(\{a_1, ..., a_{i-1}\})) \preceq (P_i \leftarrow \mathsf{hds}(\{a_1, ..., a_{n-k}\})) \preceq A_1.$$

It thus remains to consider the case $i = n+1$. From $A_1 \vdash \langle A_2, \bigwedge \vec{P}_{n+1}\rangle$ and $A_2 \preceq \mathsf{hds}(A_2)$ we get $\forall P_{n+1} \in \vec{P}_{n+1} : A_1 \vdash \langle \mathsf{hds}(A_2), P_{n+1}\rangle$, and hence by Lemma 6.1,

$$(P_{n+1} \leftarrow \mathsf{hds}(\{a_1, ..., a_n\})) \preceq A_1,$$

as required. □

6.5.6 Example

Let

$A = \{p \leftarrow q., r \leftarrow s., u \leftarrow v.\}$,
$Avail = \{\langle A', z\rangle \mid A' \subseteq A\}$,
$A_0 = \{q., s., v., z \leftarrow p, r, u.\}$,
$\mathbf{Visible}(A_0) = \emptyset$, and
$\pi_0 = \langle \emptyset, q \vee s\rangle$.

We are interested in finding if π_0 is opaque or detectable in A_0. It is clear that the only probe that is positive in A_0 is $\pi = \langle A, z\rangle$ and all other probes in *Avail* are negative. Since the queries in the available probes do not have any disjunction, $\mathbf{flatten}_{A_0}(Avail)$ contains one pair of probe sets $(\{\pi\}, Avail \backslash \{\pi\})$. Also $\mathbf{dnf}(\neg(q \vee s)) = \{(\emptyset, \{q, s\})\}$. Therefore, **Init** contains one initial state $\sigma_0 = \langle \Pi^+, \Pi^-, \emptyset\rangle$ where $\Pi^+ = \{\pi\}$ and $\Pi^- = Avail \backslash \{\pi\} \cup \{\langle \emptyset, q \vee s\rangle\}$.

Beginning for σ_0, the only possible transition is when $\widetilde{A} = A$, as for all $\widetilde{A} \subsetneq A$, $\langle \widetilde{A}, z\rangle \in \Pi^-$. For all six different permutations of $\widetilde{A}$, we have the candidates $A''_8 - A''_{13}$. The first four fail to make $\langle \emptyset, q \vee s\rangle \in \Pi^-$ negative. But $A''_{12} - A''_{13}$ pass the test. Hence, $\langle \emptyset, \Pi^-, A''_{12}\rangle$ (in the case of selecting A''_{12}) is a final state and therefore π_0 is opaque. To make A''_{12} observationally equivalent to A_0, we can easily inject the atom p_{Hi} into the bodies of the clauses in A''_{12}.

6.5.7 Soundness and completeness of the algorithm

In this section, we provide the soundness and completeness of the algorithm. The proofs are fully presented in [14].

Definition 6.13 (Reachability)**.** We write $\sigma \to \sigma'$ to denote $\sigma \xrightarrow{\pi} \sigma'$ for some $\pi \in \mathit{Prb}$ and states σ, σ'. We write σ^* for the reflexive-transitive closure of $\to$. We write $\vdash \sigma$ if $\sigma_0 \to^* \sigma$ for some $\sigma_0 \in \mathbf{Init}$.

Lemma 6.8 (Soundness)**.** If $\vdash \langle \emptyset, \Pi_0^-, A\rangle$, then π_0 is opaque in A_0.

Proof. If $\vdash \langle \emptyset, \Pi_0^-, A\rangle$, their exists $\sigma_I = \langle \Pi^+, \Pi^-, \mathsf{Visible}(A_0)\rangle \in \mathbf{Init}$ such that $\sigma_I \to^* \langle \emptyset, \Pi_0^-, A\rangle$, where Π^+, Π^- are specified as in (INIT), and $\Pi^- = \Pi_0^-$.

There is a series of (PROBE) applications starting from σ_I, with one $\xrightarrow{\pi}$ transition for each $\pi \in \Pi^+$, leading to $\langle \emptyset, \Pi_0^-, A\rangle$. Hence by repeated application of Lemma 6.6, we have $\forall \pi \in \Pi^+ : A \vdash \pi$. From the definitions of (INIT) and (PROBE) it follows that $\forall \pi \in \Pi^- : A \not\vdash \pi$, and that p_{Hi} does not occur in A. Furthermore, $\mathsf{Visible}(A_0) \subseteq A$. Therefore, by Lemma 6.5, π_0 is opaque in A_0.

□

Lemma 6.9 (Completeness)**.** If π_0 is opaque in A_0, then there exists $\Pi_0^- \subseteq \mathit{Prb}$ and $A \subseteq \mathit{Cls}$ such that

$$\vdash \langle \emptyset, \Pi_0^-, A\rangle$$

Proof. If π_0 is opaque in A_0, then by Lemma 6.5, there exists $\sigma_I = \langle \Pi_0^+, \Pi_0^-, A_I\rangle \in \mathbf{Init}$ and $A' \subseteq \mathit{Cls}$ such that $A_I \preceq A'$ and $\forall \pi \in \Pi^+ : A' \vdash \pi$ and $\forall \pi \in \Pi^- : A' \not\vdash \pi$.

By Lemma 6.3, there exists $(\Pi^+, \Pi^-) \in \mathsf{flatten}_{A_0}(\mathit{Avail})$ such that $\forall \pi \in \Pi^+ : \; A' \vdash \pi$ and $\forall \pi \in \Pi^- : \; A' \not\vdash \pi$. Furthermore, by Lemma 6.2, there exists (Π_S^+, Π_S^-) as defined in (INIT) such that $\forall \pi \in \Pi_S^+ : \; A' \vdash \pi$ and $\forall \pi \in \Pi_S^- : \; A' \not\vdash \pi$. Let $\Pi_0^+ = \Pi^+ \cup \Pi_S^+$ and $\Pi_0^- = \Pi^- \cup \Pi_S^-$. Then $\sigma_I = \langle \Pi_0^+, \Pi_0^-, \emptyset\rangle \in \mathbf{Init}$.

We now prove that for all $\Pi_1^+ \subseteq \Pi_0^+$, there exists $A \preceq A'$ such that

$$\langle \Pi_1^+, \Pi_0^-, A_I\rangle \to^* \langle \emptyset, \Pi_0^-, A\rangle.$$

The proof proceeds by induction on $m = |\Pi_1^+|$. If $m = 0$, the statement holds trivially.

If $m > 0$, there exists $\pi = \langle A_\pi, \bigwedge \vec{P}\rangle \in \Pi_1^+$. By the inductive hypothesis, there exists $A_1 \preceq A'$ such that $\langle \Pi_1^+ \setminus \{\pi\}, \Pi_0^-, A_I\rangle \to^* \langle \emptyset, \Pi_0^-, A_1\rangle$. By inspection of (PROBE), we also have $\langle \Pi_1^+, \Pi_0^-, A_I\rangle \to^* \sigma = \langle \{\pi\}, \Pi_0^-, A_1\rangle$. It remains to show that there exists $A \preceq A'$ such that $\sigma \xrightarrow{\pi} \langle \emptyset, \Pi_0^-, A\rangle$.

Since $\pi \in \Pi_0^+$, we have $A' \vdash \pi$. Then there exists a minimal $\widetilde{A}$ such that $\widetilde{A} \subseteq A_\pi$ and $A' \vdash \langle \widetilde{A}, \bigwedge \vec{P}\rangle$. Let $n = |\widetilde{A}|$. Then by Lemma 6.7, there exists $\langle a_1, ..., a_n\rangle \in \mathsf{Perm}(A_\pi)$ and there exist

$$\begin{aligned}\vec{P_i} &= \mathsf{bd}(a_i),\ \text{for } i \in \{1, ..., n\},\ \text{and}\\ \vec{P}_{n+1} &= \vec{P}\end{aligned}$$

such that for all $k \in \{1, ..., n+1\}$ and all $P_k \in \vec{P_k}$

$$(P_k \leftarrow \mathsf{hds}(\{a_1, ..., a_{k-1}\})) \preceq A'.$$

Hence $A'' = \bigcup_{k=1}^{n+1} \bigcup_{P_k \in \vec{P_k}} \{P_k \leftarrow \mathsf{hds}(\{a_1, ..., a_{k-1}\})\} \preceq A'$, and thus there exists $A = A_1 \cup A'' \preceq A'$. By anti-monotonicity of $\nvdash$, we get $\forall \pi' \in \Pi_0^- : A \nvdash \pi'$. Hence all conditions of (Probe) are satisfied.

Therefore, $\sigma_i \rightarrow^* \sigma \xrightarrow{\pi} \langle \emptyset, \Pi_0^-, A \rangle$.

□

The above lemmas allow us to prove the following soundness and completeness theorem:

Theorem 6.1 (Soundness and completeness)**.** π_0 is opaque in A_0 iff there exists $\Pi^- \subseteq \mathit{Prb}$ and $A \subseteq \mathit{Cls}$ such that $\vdash \langle \emptyset, \Pi^-, A \rangle$.

Proof. This follows directly from Lemmas 6.8 and 6.9. □

We are also able to find an upper bound for the number of transitions.

Theorem 6.2. The number of $\xrightarrow{\langle A, \bigwedge \vec{P} \rangle}$ transitions from any state is bounded by

$$\sum_{m=0}^{n} \frac{n!}{(n-m)!}$$

where $n = |A|$.

Proof. The number of $\widetilde{A} \subseteq A$ of size m is $\binom{n}{m}$, and for each such $\widetilde{A}$, there are $m!$ permutations. The size m runs from 0 to m, hence the number of transitions is bounded by

$$\sum_{m=0}^{n} m! \binom{n}{m} = \sum_{m=0}^{n} \frac{n!}{(n-m)!}$$

□

6.6 Implementation and optimizations

We implemented the algorithm based on the state transition system in figure 6.1 in F# functional programming language. We compute **Init** in lazy enumeration and then perform depth first search based on the transition rule (Probe). The back end is a Datalog engine, which computes the evaluation relation $\vdash$. The front end contains a parser for the problem which includes four blocks for A_0, $\mathbf{Visible}(A_0)$, *Avail* and π_0. The GUI displays the witness, in the case that the probe is found to be opaque. In our implementation, it is possible for the users to go through all the witnesses one after each other.

What does the feature of enumerating over the witnesses buy us? In our experiments, there were several cases when we expected a property to be detectable in a policy, but the algorithm reported opacity. When we investigated the witnesses, we found that they are all "improbable" cases. Therefore, this constructive method allows us to detect a property in a policy with a rather high likelihood.

Theorem 6.2 shows that the number of transitions and therefore, search space is very high. This makes the algorithm to be practically infeasible even for small policies. Therefore, we need to come up with implementing some optimization methods to reduce the search space.

6.6.1 Order independence

One of the important features of the algorithm that significantly reduces the search state is that the order of processing the probes in Π^+ is irrelevant. This feature reduces the search space by the order of $|\Pi^+|!$. Hence, it is sufficient to fix a particular order for the probes at the initial state.

Lemma 6.10 (Order independence)**.** If $\sigma_0 \xrightarrow{\pi_1} \sigma_1 \xrightarrow{\pi_2} \sigma_2$ then three exists σ_1' such that $\sigma_0 \xrightarrow{\pi_2} \sigma_1' \xrightarrow{\pi_1} \sigma_2$.

Proof. The proof of the lemma directly follows from the definition of (Probe). □

6.6.2 Redundant probes

If $\pi = \langle A, \varphi \rangle \in \mathit{Avail}$, it is possible for the adversary to send all the probes of the form $\{\langle A', \varphi \rangle \mid A' \subseteq A \rangle\}$. We call π a *downward closed probe* and simply mark it by a plus sign in the specification of *Avail*.

Definition 6.14. Let $\pi = \langle A, \varphi \rangle$ and $\pi' = \langle A', \varphi' \rangle$. We write $\pi \subseteq \pi'$ iff $\varphi = \varphi'$ and $A \subseteq A'$. Similarly, we write $\pi \subsetneq \pi'$ iff $\pi \subseteq \pi'$ and $\pi \neq \pi'$.

If $\pi_1 \subseteq \pi_2$, by the monotonicity of $\vdash$, it is clear that if π_1 is a positive probe, then π_2 is also positive. A similar argument exits for the probes in Π^-. We formalize this intuition by the following lemma.

Lemma 6.11. Let π_1, π_2 be two monotonic probes such that $\pi_1 \subseteq \pi_2$.

- $\exists A' : \langle \Pi^+ \cup \{\pi_1, \pi_2\}, \Pi^-, A\rangle \to^* \langle \emptyset, \Pi^-, A'\rangle$ iff
  $\exists A'' : \langle \Pi^+ \cup \{\pi_1\}, \Pi^-, A\rangle \to^* \langle \emptyset, \Pi^-, A''\rangle$ and $A'' \vdash \pi_2$.

- $\exists A' : \langle \Pi^+, \Pi^- \cup \{\pi_1, \pi_2\}, A\rangle \to^* \langle \emptyset, \Pi^- \cup \{\pi_1, \pi_2\}, A'\rangle$ iff
  $\exists A'' : \langle \Pi^+, \Pi^- \cup \{\pi_2\}, A\rangle \to^* \langle \emptyset, \Pi^- \cup \{\pi_2\}, A''\rangle$ and $A'' \nvdash \pi_1$.

Proof. The lemma is trivially true if $\pi_1 = \pi_2$. We now assume $\pi_1 \subsetneq \pi_2$.

1. ($\Rightarrow$) From the definition of (Probe) and the left hand side of the equivalence, the existence of A'', such that the first part of the right hand side holds, is clear. By Lemma 6.6, we have $A'' \vdash \pi_1$, which, by monotonicity of $\vdash$ and π_1 and π_2, implies $A'' \vdash \pi_2$.

   ($\Leftarrow$) The right hand side of the equivalence implies $\langle \Pi^+ \cup \{\pi_1, \pi_2\}, \Pi^-, A\rangle \to^* \langle \{\pi_2\}, \Pi^-, A''\rangle$, and in particular, within this chain there exists a transition $\sigma \xrightarrow{\pi_1, \vec{a}} \sigma'$ for some states $\sigma, \sigma', \vec{a}$. Since $\pi_1 \subsetneq \pi_2$, we have $\langle \{\pi_2\}, \Pi^-, A''\rangle \xrightarrow{\pi_2, \vec{a}} \langle \emptyset, \Pi^-, A'\rangle$, where $A' = A''$.

2. ($\Rightarrow$) Assuming the left hand side of the equivalence, the existence of A'', such that the first part of the right hand side holds, is clear. From the definition of (Probe), we have $A'' \nvdash \pi_2$, which, by antimonotonicity of $\nvdash$ and the monotonicity of π_1 and π_2, implies $A'' \nvdash \pi_1$.

   ($\Leftarrow$) The right hand side of the equivalence implies that $A'' \nvdash \pi_1$ and $A'' \nvdash \pi_2$, and hence $\langle \Pi^+, \Pi^- \cup \{\pi_1, \pi_2\}, A\rangle \to^* \langle \emptyset, \Pi^- \cup \{\pi_1, \pi_2\}, A'\rangle$, where $A' = A''$.

□

We use Lemma 6.11 to eliminate redundant probes from the initial states. If $\langle \Pi_0^+, \Pi_0^-, A\rangle \in$ **Init**, then it will be transformed to the state $\langle \Pi_1^+, \Pi_1^-, A\rangle$ where

- $\Pi_1^+ = \{\pi \in \Pi_0^+ \mid \neg\exists \pi' \in \Pi_0^+ : \pi' \subsetneq \pi\}$, and

- $\Pi_1^- = \{\pi \in \Pi_0^- \mid \neg\exists \pi' \in \Pi_0^- : \pi \subsetneq \pi'\}$.

6.6.3 Conflicting probes

We can discard the initial states $\sigma_0 = \langle \Pi^+, \Pi^-, A\rangle \in$ **Init** in which there exists monotonic probes $\pi_1 \in \Pi^+, \pi_2 \in \Pi^-$ such that $\pi_1 \subseteq \pi_2$. The following lemma proves that no transition from this initial state exists.

Lemma 6.12. Let π_1, π_2 be monotonic probes such that $\pi_1 \subseteq \pi_2$. There exists no state σ such that $\langle \Pi^+ \cup \{\pi_1\}, \Pi^- \{\pi_2\}, A\rangle \xrightarrow{\pi_1} \sigma$.

Proof. Suppose the contrary. Then there exists $\sigma = \langle \Pi^+, \Pi^- \cup \{\pi_2\}, A'\rangle$ with the property as stated. Moreover, $A' \vdash \pi_1$, by Lemma 6.6, and $A' \nvdash \pi_2$, by (Probe). Since $\pi_1 \subseteq \pi_2$, the former implies $A' \vdash \pi_2$, which contradicts the latter. □

6.6.4 Minimally positive probes

We first define the concept of minimally positive probes in our algorithm.

Definition 6.15 (Minimally positive)**.** A probe $\pi = \langle A, \varphi\rangle \in \mathit{Prb}$ is *minimally positive* in $A' \subseteq \mathit{Cls}$ iff φ is monotonic and $A' \vdash \pi$ and for all $\pi' \subsetneq \pi : A' \not\vdash \pi$.

We use to notation $\sigma \xrightarrow{\pi, \vec{a}} \sigma'$ to parameterize the transition with the corresponding permutation of clauses in $\widetilde{A}$ ($\vec{a} \in \mathbf{perms}(\widetilde{A})$). Lets assume that π^+ is a minimally positive probe. The definition of minimally positive shows that the only possible transition for $\sigma \xrightarrow{\pi^+, \vec{a}} \sigma'$ is when $a \in \mathbf{perms}(A)$, or in the other words $\widetilde{A} = A$.

Therefore, if a probe π^+ is marked as minimally positive probe in the set *Avail*, the rule (Probe) will be replaced with a much simpler one, when the selection of $\widetilde{A}$ is deterministic and $\widetilde{A} = A$. This optimization reduces the number of transition from a particular state by the factor of $2^{|A|}$.

6.7 Experimental results

We compared the computational time for the opacity verification algorithm for several test cases, and compared the results when applying the three different optimization methods.

6.7.1 Test cases

We performed our tests based on the delegation policy introduced in section 6.4. We derived six test cases (TC1-TC6) from the policy and measured the computation times and the number of calls to Datalog engine. Our experiments show that the verification time is directly related to the number of calls to the Datalog engine. We run our experiments in four different configurations: (1) verification without applying any optimization method (2) eliminating conflicting probes (section 6.6.3) (3) eliminating redundant probes (section 6.6.2) (4) applying both optimizations in (2) and (3) together. We performed the experiments on an Intel Core2 Dou P9500 2.53GHz workstation with 4GB RAM running windows 7 32-bit.

TC1. We consider the policy A_0 to be the basic delegation policy (clauses (1) - (7)) and $\textbf{Visible}(A_0) = \emptyset$. The adversary possesses the credentials $A_{\texttt{Eve}} = \{(8) - (11)\}$ and the query is:

$$\varphi_{\texttt{Eve}} = \texttt{canExec(Cluster,Eve,Job)}$$

The set of available probes contain 2^4 probes: $\mathit{Avail} = \{\langle A, \varphi_{\texttt{Eve}} \mid A \subseteq A_{\texttt{Eve}} \rangle\}$. We are interested weather `Eve` can detect if `Bob` is a member of `Cluster`. The input probe π_0 is specified as

$$\pi_0 = \langle \emptyset, \neg\texttt{isMem(Cluster,Bob)} \rangle$$

which allows us to conclude $A_0 \vdash \neg\texttt{isMem(Cluster,Bob)}$. The probe π_0 is detectable in A_0.

TC2. We add the atomic clause `isMem(Cluster,Bob)` to A_0. Therefore, `Bob` is now a member of `Cluster`. We change π_0 to $\langle \emptyset, \texttt{isMem(Cluster,Bob)} \rangle$. Our experiments show that the probe containing the clauses (8), (9) and (11) is positive, while the probe containing the clauses (8) and (9) is negative. Therefore, the probe (11) is relevant. It is only possible if its body atom `isMem(Cluster,Bob)` is derivable.

The tool reports the probe π_0 is (correctly) opaque. In the witnesses, there is a clause which is rather unlikely:

```
isMem(Cluster,Bob)←
   isMem(CA,Eve), owns(CA,Eve,Job).
```

Therefore, we can conclude that `isMem(Cluster,Bob)` in the policy with a high probability.

TC3. We add three irrelevant clauses $\{p_1., p_2., p_3.\}$ to $A_{\texttt{Eve}}$ in order to increase the number of probes in *Avail* by the factor of 8.

TC4. We manually prune the set *Avail* in TC3 in order to build the sufficient set of probes for detecting π_0:

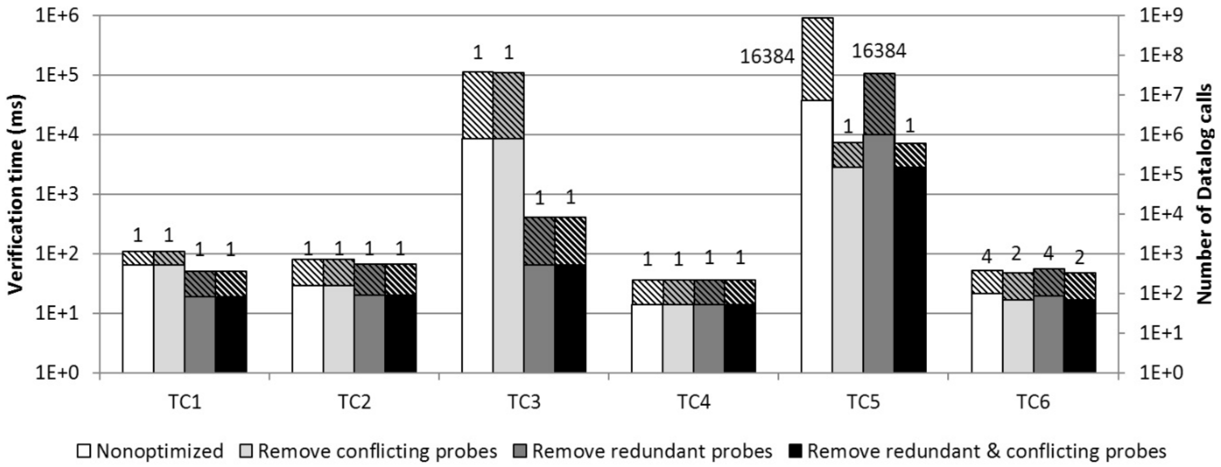

Figure 6.2: Verification time (striped) and number of calls to the Datalog engine (plain) on log scales. Numbers above bars denote the number of initial states.

$\langle \{(8), (9), (10)\}, \varphi_{\texttt{Eve}} \rangle^*$

$\langle \{(8), (9), (11), p_1., p_2., p_3.\}, \varphi_{\texttt{Eve}} \rangle$

$\langle \{(9), (10), (11), p_1., p_2., p_3.\}, \varphi_{\texttt{Eve}} \rangle$

$\langle \{(8), (10), (11), p_1., p_2., p_3.\}, \varphi_{\texttt{Eve}} \rangle$

Only the first of the above probes is positive. The structure of the probes 2-4 allow us to select the first probe as the minimally positive, which makes the verification process more efficient.

TC5. We use the same policy as in TC1, but change the query to

$\varphi_{\texttt{Eve}} = \texttt{canExec(Cluster,Eve,Job)} \wedge \neg\texttt{isBanned(Cluster,Eve)}.$

π_0 is still detectable in A_0.

TC6. We prune the probes in TC5 to the sufficient set for proving detectability:

$\langle \{(8), (9), (10)\}, \varphi_{\texttt{Eve}} \rangle$

$\langle \{(8), (9)\}, \varphi_{\texttt{Eve}} \rangle$

$\langle \{(8), (9), (11)\}, \varphi_{\texttt{Eve}} \rangle$

From the above probes, only the first one is positive. As $\varphi_{\texttt{Eve}}$ is not monotonic, we can not use the minimally positive probe optimization.

The query $\varphi_{\texttt{Eve}}$ in TC1-TC4 does not contain negation and therefore, no conflicting probe exists after flattening process. This is clearly shown in figure 6.2 that conflicting probe optimization does not have any effect on verification time and DATALOG engine calls.

TC2 runs 40% faster than TC1. This is because the property in TC1 is detectable. Therefore, the program traverses all the possible states to prove its detectability. For TC2, the program stops when it finds the first witness.

TC3 shows how the number of available probes dramatically increases the verification time (here by the factor of 8) when no optimization is used. Optimization increases the performance by the factor of 8.

The most effective strategy to decrease the verification time is manually picking the relevant probes, or in the other words, manually pruning the set of available probes. TC4 and TC5 are the cases that demonstrate such an intuition. TC4 decreases the time by the factor of 3,150 compared to TC3 in non-optimized configuration. TC6 decreases the time by the factor of 19,000 compared to non-optimized TC5 and 150 non-optimized configurations.

TC5 shows for non-monotonic probes, applying the conflicting probes optimization significantly reduces the number of initial states (in our case, from 16,384 to only 1). This will result in considerable increase in performance.

Our last experiment demonstrates the effect of the size of *Avail* in verification time. We created two set of available probes, one created by adding a number of trivially positive probes, and the other one by adding negative ones. The verification time increases exponentially when the number of positive probes increases (which is predictable, as it increases the number of states), but increases linearly when the number of negative probes increases. This is also expected, as the negative probes do not result in branching.

6.8 Summary

In this research, we first proposed a general framework of probing attacks, and formalized the notions of policy, probe and adversary. We instantiated our framework into DATALOG, which is the basis of many existing policy languages.

The main contribution of the research is the answer for the following *open* question:

Is the problem of opacity in DATALOG policies decidable?

We answered this question in the positive by presenting a complete decision procedure for opacity. The algorithm tries to find the witnesses to prove the opacity of a property (or in general, a probe) in the policy. The witnesses masquerade the original policy the way that they behave the same against submitted probes, but the property is not true in them.

We also provide the opportunity to consider *possibilistic* information flow in a policy. The algorithm and the implementation is designed in a way that it can enumerate over all possible witnesses for the opacity. The adversary can verify the possibility that a witness contain realistic assertions. Therefore, with a specific probability, an adversary can detect a property in a policy, which is reported as opaque.

The algorithm has two limitations: (1) the set of available probes contain only ground probes. Although it is a realistic assumption, there may be some cases where the attacker self-issues non-ground credentials. (2) The set of available probes is finite. As before, this assumption is realistic in many cases, but it should be useful to prove the opacity in the cases that adversary has access to infinitely many probes. We consider solving the above limitations as the future work.

CHAPTER 7

CONCLUSION

Confidentiality of sensitive data and prevention of unauthorized access to the resources is one of the main concerns in multi-agent collaborative systems. Access control is the mechanism to enforce the security requirements for accessing information in the system. To provide the assurance that the security requirements are correctly enforced, the system should be evaluated against required properties.

In this thesis, we developed several techniques that enable us to investigate temporal-epistemic properties of access control systems in an automated way. The key contributions are:

- We implemented a model-checking method in order to verify properties of access control systems considering the knowledge of the coalition of agents which is gained by reading system information. The output of this research is implemented as a verification tool called PoliVer [66, 65] which comparing to the similar verification framework RW [94] has increased verification time and memory efficiency. This improvement is achieved by replacing the knowledge states in RW (which are used to introduce a memoryful approach) with system states. Although PoliVer does not retain memory (history of reading), extra variables for storing the history of reading information can be simply incorporated into the policy when it is required.

- The abstraction of the knowledge in PoliVer increases the efficiency of model-checking while there is still a category of information leakage vulnerabilities that PoliVer is not able to detect. Therefore, we introduce a complimentary framework in order to identify information leakage as a result of reasoning in dynamic policies which is based on the interpreted systems. To increase the efficiency, we developed a counterexample-guided abstraction refinement technique for the verification of temporal and epistemic properties.

- We finally and as a research on stateless policies, proposed a sound and complete method for detecting information leakage in DATALOG-based policies [15, 16].

This chapter discusses the outcomes of this research and future work.

7.1 Summary

We have divided the research of temporal-epistemic evaluation of *dynamic* access control policies into two major parts: first, we estimate knowledge with the information that is gained by reading system information, and second, we study the verification of knowledge which is gained by *reasoning*.

7.1.1 PoliVer

We developed PoliVer based on the model-checking method we proposed for the verification of temporal-epistemic properties in chapter 4. Given a set of rules, initial condition and a goal, PoliVer checks if a coalition of agents can achieve the goal through a finite sequence of actions. The specifications of PoliVer are categorized as follows:

- **Expressive policy language:** Action rules and read permission rules in PoliVer policy language can express the laws in a wide variety of access control systems. Compared to RW [94, 95], the policy language allows updating a group of propositional variables when executing an action. The permission for an action allows quantification and negation and therefore is flexible enough to express separation of duty, mutual exclusion between the roles and role inheritance. *Variable bulk update* plays an important role to support integrity of constraints.

- **Query language:** The query asks if a set of agents can collaborate to achieve a goal, defined as a property, through a sequence of actions beginning from a set of initial states. The query language is also flexible from the point of collaborative goals, temporal properties and epistemic properties in the form of reading system propositions. For instance, a goal that contains $\langle \texttt{review}(\texttt{p}, \texttt{b}) \rangle$ looks if the coalition in the goal can reach to a state that at least one of them is able to read the value of $\texttt{review}(\texttt{p}, \texttt{b})$. Nested goals are also supported by the query language.

- **Model-checking algorithm:** The model-checking formalism of PoliVer finds the reachability of a goal, together with evaluating the knowledge of the agents over the information they require to know in order to be able to achieve the goal. The initial knowledge of the coalition is determined in the query and the knowledge in each state is the accumulation of the knowledge gained by performing actions or reading system information along the strategy.

In general, PoliVer verifies epistemic properties by approximating the knowledge by readability. Our experimental results show that the time and memory efficiency increases by this approximation, which is sufficient to detect a high percentage of vulnerabilities.

7.1.2 Reasoning about knowledge in access control systems

To cover the evaluation of information leakage vulnerability that can not be detected by PoliVer, this thesis introduces a complimentary formalization based on interpreted systems. Using interpreted systems allows us to verify the knowledge expressed by the modal logic KT45^n. We have shown that there are some information leakage vulnerabilities that can not be detected by the state of art verification tools line RW, DynPAL [11] and PoliVer. The major obstacle of modelling an access control system using interpreted systems framework is the state explosion problem. In order to make the verification more efficient, we introduced a fully automated abstraction and refinement technique when the property is ACTLK. The method is an extension of CEGAR (Counterexample guided abstraction refinement) [30]. Using this method, we are able to check a tree-like counterexample generated by the model-checker in order to find if it is also valid in the concrete model. While the original CEGAR only supports linear counterexamples produced by verifying temporal properties, our method covers tree-like counterexamples produced by verifying temporal-epistemic properties. The method is implemented in Microsoft F# and uses the model-checker for multi-agent systems MCMAS as the model-checking engine.

Some important safety properties that include the negation of knowledge modality do not fit into the category of ACTLK. For such cases, our tool uses an interactive refinement which is described in chapter 5. Therefore, it is valuable to automate abstraction refinement such for such safety properties together with appropriate refinement heuristics.

7.1.3 Information leakage in Datalog-based policies

In the category of stateless policies and as an independent research in Microsoft Research Cambridge, we looked at information leakage vulnerability of Datalog-based credential systems. The problem of opacity in Datalog-based policies was an open problem [12]. In our research, we proposed a sound, complete and terminating algorithm that given a set of available probes (refer to chapter 6), it is able to decide whether a property is opaque in the policy or detectable. The algorithm uses the concept of observational equivalence similar to the one we used for dynamic policies. When a property is found to be opaque, our implementation allows the security analyst to traverse through all the witnesses. This may informally judge the likelihood of opacity of a property as some witnesses may be far from a practically real policy.

7.2 Future work

In this thesis, we demonstrated the effectiveness and efficiency of the developed techniques using small size case studies that were used by other researchers. But it is very important to evaluate large size real systems like Facebook, EasyChair and Google docs, which we will consider for the future work. Evaluating such systems also requires overcoming the limitations of the model-checker for multi-agent systems to handle large scale models.

In the general case, Access control systems can have unbounded numbers of states, since objects can be created dynamically. Unfortunately, unbounded state model-checking in general is undecidable, therefore we can model-check only finite systems. To address this problem, in this thesis and for evaluating dynamic policies, we adopt the "small scope hypothesis" defined by the authors of Alloy [61] that is suitable for finding a high proportion of errors and can be expanded to the large model. Also experimental results show that a sufficient scope can be found in order to provide the confidence of having no bug in the system [6]. So, by selecting a sufficient scope, we are able to simulate an access control system with unbounded numbers of states with a finite state one and verify the required properties over it.

But still evaluating a system with unbounded number of objects is valuable and more promising. To study the evaluation of access control systems for unbounded number of objects through model-checking, we need to apply the appropriate abstraction techniques. One of the techniques that can possibly help in achieving such a goal is *symmetry reduction technique*. Using such technique, we will be able to remove the agents that behave in a similar way, which may be potentially infinite. Therefore, the original model will be reduced to a finite state model and model-checking problem becomes decidable.

Furthermore and as we discussed before, another work that is left for the future in this thesis is the appropriate heuristics for abstraction refinement when verifying the properties with the negation of knowledge modality. Improving the heuristics for the cases that we have more than one candidate for the refinement is also an interesting work for the future.

Appendices

APPENDIX A

CALCULATING BDD-BASED TRANSITION FUNCTION IN POLIVER

In section 4.2, the pre-states are defined as a set of states that satisfy the properties in the definition 4.5. For the implementation, we need to find the transition function based on binary decision diagram presentation.

A.1 An overview of binary decision diagrams

Binary decision diagrams are a method of representing *Boolean functions*. Boolean functions are the functions with the arguments of type *Boolean variables*, and return a Boolean value. BDDs are a class of *binary decision trees* where the non-terminal nodes represent Boolean variables and terminal nodes are Boolean values (0 and 1). Each binary decision tree represents a Boolean function of the variables that appear as non-terminal nodes. BDDs are capable of getting compact by removing of duplicated terminal nodes, and removing redundant and duplicate non-terminals. We call these compact diagrams as *reduced* BDDs.

Ordered binary decision diagrams (OBDD) are the diagrams that are imposed by a variable ordering. The ordering prevents a variable to occur several times along a path. Another important feature of OBDDs is that the reduced OBDD (denoted by ROBDD) that represents a Boolean function is unique.

Several algorithms are defined for ROBDDs: the algorithm `reduce` applies the reduction rules over a BDD. The algorithm `apply` implements the operations $\wedge$, $\vee$ and $\oplus$ (XOR) on binary decision diagrams. If B_f is a BDD representing formula f, then $\texttt{restrict}(0, x, B_f)$ is equivalent to $f[0/x]$ and $\texttt{restrict}(1, x, B_f)$ is equivalent to $f[1/x]$. The algorithm `exists` removes the constraints on a subset of variables. If f is a Boolean formula, then $\exists.f$ is defined as $f[0/x] \vee f[1/x]$. Therefore, the algorithm `exists` is equivalent to:

$$\texttt{apply}(\vee, \texttt{restrict}(0, x, B_f), \texttt{restrict}(1, x, B_f))$$

In *symbolic model-checking*, sets of states are presented by BDDs. In a transition system, each state is assigned by a vector of Boolean variables. Note that in our policy verification method, the states are represented by the valuation of system propositions, which are Boolean variables by themselves.

In CTL model-checking and in our approach, we need set operations like intersection, union and complementation. These set operations are equivalent to $\wedge$, $\vee$ and $\neg$ in BDD operations using the algorithm `apply` (note that negation is implementable via $\oplus 1$). The core function that we use in our algorithm is $PRE^{\exists}_{\alpha}(X)$. This function is a special case of the standard function $\text{pre}_{\exists}(X)$ which takes a subset X of states and returns all the states that have transition to some states in X. $PRE^{\exists}_{a}$ is dedicated for the transitions made by performing action α in our action-based transition system.

Let $\hat{p}$ denote the vector of Boolean variables $(p_1, \ldots, p_n)$. If $B_{\rightarrow}$ is the OBDD representing transition relation, B_X is the OBDD for the set of states X, and primed version of the variables denote the variables in the successor states in the transitions, then the OBDD $\texttt{exists}(\hat{p}', \texttt{apply}(\wedge, B_{\rightarrow}, B_{X'}))$ computes the $\text{pre}_{\exists}(X)$ where $B_{X'}$ is the OBDD of B_X where all the variables are replaced with their primed versions.

A.2 Transition relation calculation

We use the conventional method of finding pre-states in order to calculate $PRE^{\exists}_{a}$. We represent the Boolean formula to show how the general calculation works. This formula will be encoded to OBDD in implementation phase.

Let action a be defined as $\alpha : \varepsilon \leftarrow \ell$ and X be a set of states. Consider $f_{PRE^{\exists}_{\alpha}(X)}$ as the formula satisfying $PRE^{\exists}_{\alpha}(X)$ and $\{p_1, \ldots, p_n\} = \textbf{prop}(f_X)$. $f_{PRE^{\exists}_{\alpha}(X)}$ can be calculated by the following method:

$$f_{PRE^{\exists}_{\alpha}(X)} = \exists p'_1 \ldots \exists p'_n . (\theta_\alpha \wedge f_X[p'_i/p_i : 1 \leq i \leq n])$$

$f_X[p'_i/p_i : 1 \leq i \leq n]$ is the primed version of the formula representing the set of states X and θ_α is the transition formula defined as:

$$\theta_\alpha = \Big(\bigwedge_{p_i \in \textbf{effect}_+(\alpha)} p'_i \Big) \wedge \Big(\bigwedge_{q_i \in \textbf{effect}_-(\alpha)} \neg q'_i \Big) \wedge \Big(\bigwedge_{r \in \textbf{prop}(f_X) \setminus \textbf{effect}(\alpha)} r \leftrightarrow r' \Big) \wedge \ell \wedge \bigwedge_{l_i \in Lit^*} l_i$$

where Lit^* be the set of literals that are tagged by $*$ in the query. The last conjunction dictates the constraint of fixed literals in the query.

APPENDIX B

CALCULATING FORWARD BDD-BASED TRANSITION FUNCTION

In chapter 5 section 5.3, we introduced a symbolic transition function for forward traversal of the paths in the counterexamples returned by the model-checker. For the implementation, we need to find the transition function based on binary decision diagram presentation.

If ψ is a formula, then ψ' denotes the primed version of ψ if all the propositions in ψ are substituted with the primed ones ($\psi' = \psi[p'_i/p_i : p_i \in \mathbf{prop}(\psi)]$). When performing the action $\alpha : \varepsilon \leftarrow \ell$ in the states st_ϕ (using the same notation as in chapter 4), the transition only performs on the states of $st_\phi \cap st_\ell = st_{\phi\wedge\ell}$. In the resulting states, the propositions that do not appear in ε remain the same as in the states that the transition begins. If $\{p_1, \ldots, p_n\} = \mathbf{prop}(\phi \wedge \ell)$, the transition function Θ_α is calculated in the following way:

$$\Theta_\alpha(st_\phi) = st_{\exists p'_1,\ldots,p'_n.\ \eta(\alpha,\phi)}$$

where

$$\eta(\alpha,\phi) = (\phi \wedge \ell)' \wedge (\bigwedge_{+p\in\varepsilon} p) \wedge (\bigwedge_{-q\in\varepsilon} \neg q) \wedge (\bigwedge_{r\in\mathbf{prop}(\phi\wedge\ell), \pm r\notin\varepsilon} r \leftrightarrow r')$$

APPENDIX C

LABELLED TRANSITION SYSTEMS AND INTERPRETED SYSTEMS

In this appendix, we show that the labelled transition system described in chapter 4 is a special case of interpreted systems.

Consider that $M_{\mathcal{C}} = \langle S^{lts}, Act^{lts}, S_0^{lts}, P^{lts}, \tau^{lts}, \gamma^{lts} \rangle$ is a labelled transition system derived from policy $\mathcal{C}$ with Σ_{Agent} as the set of agents. Given $\Omega = e \cup \Sigma_{\text{Agent}}$, we define a special case of interpreted systems $I^M = \langle (L_i^M)_{i \in \Omega}, (P_i^M)_{i \in \Omega}, (ACT_i^M)_{i \in \Omega}, S_0^M, \tau^M, \gamma^M \rangle$ where

- $L_e^M = S^{lts}$ and $L_i^M = \{l\}$ for all $i \in \Sigma_{\text{Agent}}$ where l is a single local state, and the set of states S^M is the Cartesian product of the local states

- $S_0^M = \{(s, l, \ldots, l) \mid s \in S_0^{lts}\}$

- For all $s^M = (s_e^M, l, \ldots, l) \in S^M, p \in P^{lts} : \gamma^M(s^M, p) = \gamma_e^M(s_e^M, p) = \gamma^{lts}(s_e^M, p)$

- $ACT_e^M = \{\Lambda\}$ and $ACT_i^M = \{\alpha \in Act^{lts} \mid \mathbf{Ag}(\alpha) = i\}$ for all $i \in \Sigma_{\text{Agent}}$. The joint action ACT^M is the set of any Ω-tuple in the Cartesian product of the local actions with only one non-Λ action

- $P_i^M(l_i^M) = ACT_i^M$ for all $i \in \Omega$ and $l_i^M \in L_i$

- For all $s^M \in S^M$ and $\alpha^M \in ACT^M$, if $s = l_e^M(s^M)$ and α is the non-Λ element of α^M, then $\tau^M(s^M, \alpha^M)$ is defined if $\tau^{lts}(s, \alpha)$ is defined, and $\tau^M(s^M, \alpha^M) = (\tau^{lts}(s, \alpha), l, \ldots, l)$.

Given policy $\mathcal{C}$ and the query $init \rightarrow \phi$ as defined in section 4.1.2, we define the satisfaction relation for interpreted system I as follows:

$(I, s, C) \models p \qquad\qquad\qquad \Leftrightarrow \quad \gamma_e(l_e(s), p) = \top$

$(I, s, C) \models \neg\phi \qquad\qquad\quad\ \Leftrightarrow \quad (I, s, C) \not\models \phi$

$(I, s, C) \models \phi_1 \vee \phi_2 \qquad\quad \Leftrightarrow \quad (I, s, C) \models \phi_1$ or $(I, s, C) \models \phi_2$

$(I, s, C) \models \langle p \rangle \qquad\qquad\quad \Leftrightarrow$ there exists a read permission $\rho : p \leftarrow \ell \in \mathcal{C}$

such that $\mathbf{Ag}(\rho) \in C$ and $(I_{\mathcal{C}}, s, C) \models \ell$

$(I, s, C) \models C' : (\phi) \qquad\qquad \Leftrightarrow$

there exists a path $s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $s = s_1$ and

(1) For all $1 \leq i < n$: $\mathbf{Ag}(\alpha_i) \in C'$

(2) For all $1 \leq i \leq n$: $(I_{\mathcal{C}}, s_i, C') \models p$ if $p* \in \mathit{init}$ and $(I, s_i, C') \not\models p$ if $\neg p* \in \mathit{init}$

(3) $(I, s_n, C') \models \phi$

$(I, s, C) \models C' : (\phi_1\ \texttt{THEN}\ \phi_2) \quad \Leftrightarrow$

there exists a path $s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $s = s_1$ and

(1) For all $1 \leq i < n$: $\mathbf{Ag}(\alpha_i) \in C'$

(2) For all $1 \leq i \leq n$: $(I, s_i, C') \models p$ if $p* \in \mathit{init}$ and $(I, s_i, C') \not\models p$ if $\neg p* \in \mathit{init}$

(3) $(I, s_n, C') \models \phi_1$ and $(I, s_n, C') \models \phi_2$

We use the notation $I \models g$ if for all $s_0 \in S_0 :\ (I, s_0, \emptyset) \models g$.

Lemma C.1. Given the query $\mathit{init} \rightarrow g$, let $I_{\mathcal{C}}$ be an interpreted system derived from policy $\mathcal{C}$ as in definition 5.7 where the set of initial states contains all the states with the environment local state defined by *init*. If $M_{\mathcal{C}}$ is the labelled transition system derived from $\mathcal{C}$ with the local states defined by *init*, then $I_{\mathcal{C}} \models g$ iff $I^M \models g$.

Proof. We first prove that for all $s \in S$ and $s_M \in S^M$ where $l_e(s) = l_e^M(s^M)$, if C is a coalition of agents, then $(I_{\mathcal{C}}, s, C) \models g$ iff $(I^M, s^M, C) \models g$.

The set of propositions P^{lts} in $M_{\mathcal{C}}$ contains all the policy propositions and therefore $P^{lts} = \Phi_{\mathcal{C}} = \Phi_e$ (see definition 5.7). So, as the local states are specified by the valuation of the local propositions, the relation $l_e(s) = l_e^M(s^M)$ implies $\gamma_e(l_e(s), p) = \gamma_e^M(l_e^M(s^M), p)$. If $p \in \Phi_e$ (g is defined over Φ_e), then the proof proceeds by structural induction over the structure of g.

Base case:

- ($\Rightarrow$) $(I_{\mathcal{C}}, s, C) \models p$ iff $(I^M, s^M, C) \models p$ From $(I_{\mathcal{C}}, s, C) \models p$ we have $\gamma_e(l_e(s), p) = \top$. By $l_e(s) = l_e^M(s^M)$ we have $\gamma_e^M(l_e^M(s^M), p) = \top$ which implies $(I^M, s^M, C) \models p$.

  ($\Leftarrow$) Similar.

Inductive cases:

Assume that $(I_{\mathcal{C}}, s, C) \models \varphi$ iff $(I^M, s^M, C) \models \varphi$ for all $s \in S$ and $s_M \in S^M$ where $l_e(s) = l_e^M(s^M)$. Also assume the same for φ_1 and φ_2.

- The proof is trivial for the cases $\neg\varphi$ and $\varphi_1 \vee \varphi_2$ and $\langle p \rangle$.

- ($\Rightarrow$) Assume that $(I_{\mathcal{C}}, s, C) \models C' : (\varphi)$. We prove that $(I^M, s^M, C) \models C' : (\varphi)$ where $l_e(s) = l_e^M(s^M)$.

  By the definition of satisfaction relation, there exists a path $s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $s = s_1$ where the conditions (1) - (3) hold. We first prove that given the above path, there exists a path $s_1^M \xrightarrow{\alpha_1^M} \ldots \xrightarrow{\alpha_{n-1}^M} s_n^M$ such that $s^M = s_1^M$ and $l_e(s_n) = l_e^M(s_n^M)$, and the path complies with the conditions (1) and (2).

  Let us consider the labelled transition system $M_{\mathcal{C}}$. Let $s_1^{lts} = l_e^M(s_1^M)$ which implies $s_1^{lts} = l_e(s_1)$. We start from the first transition along the path, which is $s_1 \xrightarrow{\alpha_1} s_2$. By procedure 3, action α_1 is one of the actions generated from some action α_1^{lts} in policy $\mathcal{C}$ where

  - $\mathbf{Ag}(\alpha_1) = \mathbf{Ag}(\alpha_1^{lts})$,
  - If the permission of α_1 holds in $l_e(s_1)$, then the permission of α_1^{lts} also holds on $l_e(s_1)$ (note that action permissions in the policy are defined over policy propositions which are the same as environment propositions in $I_{\mathcal{C}}$), and
  - α_1 updates the environment propositions in the same way as in α_1^{lts}.

  By item 2 and $s_1^{lts} = l_e(s_1)$, the permission of α_1^{lts} holds on s_1^{lts}. By item 3, s_1^{lts} evolves to a state s_2^{lts} where the values of the propositions are the same as in $l_e(s_2)$ (in the other words, $s_2^{lts} = l_e(s_2)$) if the transition is allowed.

  The transition $s_1^{lts} \xrightarrow{\alpha_1^{lts}} s_2^{lts}$ has another requirement which states that $(M_{\mathcal{C}}, s_2^{lts}, C) \models p$ if $p* \in \mathit{init}$ and $(M_{\mathcal{C}}, s_2^{lts}, C) \not\models p$ if $\neg p* \in \mathit{init}$. This follows by condition (2) in satisfaction relation for $I_{\mathcal{C}}$ and the base case.

  The path $s_1^{lts} \xrightarrow{\alpha_1^{lts}} \ldots \xrightarrow{\alpha_{n-1}^{lts}} s_n^{lts}$ can be constructed inductively using the same procedure. By the same discussion we have $s_n^{lts} = l_e(s_n)$. This proves the existence of a path $s_1^M \xrightarrow{\alpha_1^M} \ldots \xrightarrow{\alpha_{n-1}^M} s_n^M$ in I^M where $s_i^M = (s_i^{lts}, l, \ldots, l)$, and α_i^M is an Ω-tuple with $\mathbf{Ag}(\alpha_i^{lts})$-th element to be α_i^{lts} and the rest are Λ, which satisfies the conditions (1) and (2) in satisfaction relation. Condition (3) holds by inductive hypothesis.

  ($\Leftarrow$) Assume that $(I^M, s^M, C) \models C' : (\varphi)$. We prove that $(I_{\mathcal{C}}, s, C) \models C' : (\varphi)$ where $l_e(s) = l_e^M(s^M)$.

By the definition of satisfaction relation, there exists a path $s_1^M \xrightarrow{\alpha_1^M} \dots \xrightarrow{\alpha_{n-1}^M} s_n^M$ in I^M such that $s^M = s_1^M$ where the conditions (1) - (3) hold. We prove that given the above path, there exists a path $s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n$ in $I_{\mathcal{C}}$ such that $s = s_1$ and $l_e(s_n) = l_e^M(s_n^M)$, and the path complies with the conditions (1) and (2).

We start from the first transition $s_1^M \xrightarrow{\alpha_1^M} s_2^M$. Assume that policy action α_1^{lts} is the non-Λ element of α_1^M. Procedure 3 generates a set of actions from α_1^{lts} that will be used in constructing the joint actions in $I_{\mathcal{C}}$. The actions are generated from α_1^{lts} in a tree-like manner. Starting from α_1^{lts} as the root, each node has one branch (when the proposition to be read and its read permission are not affected by the action), two branches (when the proposition to be read is affected by the action) or three branches (when only the read permission is affected by the action). In the case of one branch, action remains unchanged. In the case of three branches (also applicable to the cases with two branches), if $\alpha : \varepsilon \leftarrow \ell$ is the parent and $\alpha_{c1} : \varepsilon_{c1} \leftarrow \ell_{c1}$, $\alpha_{c2} : \varepsilon_{c2} \leftarrow \ell_{c2}$ and $\alpha_{c3} : \varepsilon_{c3} \leftarrow \ell_{c3}$ are the children, then $\ell_{c1} \vee \ell_{c2} \vee \ell_{c3} \equiv \ell$. Moreover, pairwise conjunctions of ℓ_{c1}, ℓ_{c2} and ℓ_{c3} are equivalent to $\bot$. Hence, the permission of exactly one of the child actions holds in $l_e(s_1^M)$, and as the permissions are independent of other agents' local states, the permission of that child holds in any global state s_1 where $l_e(s_1) = l_e^M(s_1^M)$. As $\varepsilon_1, \varepsilon_2, \varepsilon_3 \subseteq \varepsilon$, the children update the environment local variables the same as α.

So, there exists an action α_1 in $I_{\mathcal{C}}$ where the $\mathbf{Ag}(\alpha_1^{lts})$-th element is one of leaf actions generated form α_1^{lts} in which the permission holds in s_1. Moreover, if the transition $s_1 \xrightarrow{\alpha_1} s_2$ is allowed, then $l_e(s_2) = l_e^M(s_2^M)$.

It is also trivial to show that the last requirement for the transition $s_1 \xrightarrow{\alpha_1} s_2$, which is condition (2) in satisfaction relation, also holds. Therefore the path $s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n$ can be constructed inductively using the same procedure, where $l_e(s_n) = l_e^M(s_n^M)$ and the path complies with the conditions (1) and (2). Condition (3) holds by inductive hypothesis.

- The proof for $(I_{\mathcal{C}}, s, C) \models C' : (\varphi_1 \texttt{ THEN } \varphi_2)$ iff $(I^M, s^M, C) \models C' : (\varphi_1 \texttt{ THEN } \varphi_2)$ is similar to the proof for the previous item.

As the environment initial local states in both $I_{\mathcal{C}}$ and I^M are defined by the same rule, for all $s_0 \in S_0$ there exists $s_0^M \in S_0^M$ where $l_e(s_0) = l_e^M(s_0^M)$, and vice versa. Since satisfaction of goal g in its general form is independent of the coalition parameter in the left hand side of the satisfaction relation, then for all s_0 and s_0^M where $l_e(s_0) = l_e^M(s_0^M)$ we have $(I_{\mathcal{C}}, s_0, \emptyset) \models g$ iff $(I^M, s_0^M, \emptyset) \models g$, which implies $I_{\mathcal{C}} \models g$ iff $I^M \models g$.

□

LIST OF REFERENCES

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:706–734, 1993.

[2] R. Alur, L. de Alfaro, R. Grosu, T. A. Henzinger, M. Kang, R. Majumdar, F. Mang, C. M. Kirsch, and B. Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proceedings of the 23rd international conference on software engineering*, 2001.

[3] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV 1998: Proceedings of International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998.

[4] Rajeev Alur, Radu Grosu, and Bow-Yaw Wang. Automated refinement checking for asynchronous processes. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, pages 55–72. Springer-Verlag, 2000.

[5] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[6] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the“small scope hypothesis”. In *POPL 2002: Proceedings of the 29th ACM Symposium on the principles of programming languages*, 2002.

[7] Guillaume Aucher, Guido Boella, and Leendert van der Torre. Privacy policies with modal logic: The dynamic turn. In *Deontic Logic in Computer Science*, pages 196–213, 2010.

[8] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on*

Model checking of software, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[9] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[10] Moritz Y. Becker. *Cassandra: Flexible trust management and its application to electronic health records*. PhD thesis, Computer Laboratory, University of Cambridge, 2005.

[11] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *Proceedings of 22nd IEEE Computer Security Foundations Symposium (CSF)*, 2009.

[12] Moritz Y. Becker. Information flow in credential systems. *Computer Security Foundations Symposium, IEEE*, 2010.

[13] Moritz Y. Becker, Andrew D. Gordon, and Cdric Fournet. SecPAL: Design and semantics of a decentralised authorisation language. Technical report, Microsoft Research, Cambridge, September 2006.

[14] Moritz Y. Becker and Masoud Koleini. Information leakage in Datalog-based trust management systems. Technical report, Microsoft Research, 2010.

[15] Moritz Y. Becker and Masoud Koleini. Opacity analysis in trust management systems. In *14th Information Security Conference (ISC 2011)*, 2011.

[16] Moritz Y. Becker and Masoud Koleini. Opacity analysis in trust management systems. Technical report, Microsoft Research, 2011.

[17] M.Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 139 – 154, 2004.

[18] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical report, Mitre Corporation, 1976.

[19] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. IETF RFC 2704. September 1999.

[20] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Proceedings of Security Protocols: 6th International Workshop (Position Paper)*, pages 59–63, 1998.

[21] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[22] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *Proceedings of the Second International Conference on Financial Cryptography*, pages 254–274. Springer-Verlag, 1998.

[23] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[24] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, pages 428–439, 1990.

[25] Sagar Chaki, Edmund Clarke, and Alex Groce. Modular verification of software components in C. In *IEEE Transactions on Software Engineering*, pages 385–395, 2003.

[26] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and O Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In *CAV 2002: Proceedings of International Conference on Computer-Aided Verification*, pages 359–364. Springer, 2002.

[27] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 1999.

[28] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, London, UK, 1982. Springer-Verlag.

[29] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, April 1986.

[30] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[31] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[32] Edmund M. Clarke, Yuan Lu, Broadcom Com, Helmut Veith, and Somesh Jha. Tree-like counterexamples in model checking. In *LICS 2002: Proceedings of the 17 th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002.

[33] Mika Cohen, Mads Dam, Alessio Lomuscio, and Francesco Russo. Abstraction in model checking multi-agent systems. In *AAMAS 2009: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 945–952, 2009.

[34] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM Press, New York, NY, 1977.

[35] Jason Crampton and George Loizou. Administrative scope and role hierarchy operations. In *SACMAT 2002: Proceedings of Seventh ACM Symposium on Access Control Models and Technologies*, pages 145–154, 2002.

[36] Jason Crampton and George Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security*, 6:201–231, May 2003.

[37] Frédéric Cuppens and Robert Demolombe. A modal logical framework for security policies. In *ISMIS 1997: Proceedings of the 10th International Symposium on Foundations of Intelligent Systems*, volume 1325 of *Lecture Notes in Computer Science*, pages 579–589. Springer-Verlag, 1997.

[38] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33:397–421, April 2003.

[39] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[40] Francien Dechesne, Simona Orzan, and Yanjing Wang. Refinement of kripke models for dynamics. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 111–125. Springer-Verlag, 2008.

[41] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105 – 113, 2002.

[42] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *of Lecture Notes in Computer Science*, volume 4130, pages 632–646. Springer, 2006.

[43] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181. Springer-Verlag, 1980.

[44] Constantin Enea and Catalin Dima. Abstractions of multi-agent systems. In *CEEMAS 2007: Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications*, pages 11–21. Springer-Verlag, 2007.

[45] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge.* MIT Press, Cambridge, 1995.

[46] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardis. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[47] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, 1992.

[48] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE 2005: Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM, 2005.

[49] Maria Fox and Derek Long. Pddl2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[50] Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *CSF 2008: IEEE Computer Security Foundations Symposium*, pages 149 –162, 2008.

[51] Yuri Gurevich and Itay Neeman. DKAL 2 — a simplified and improved authorization language. Technical report, Microsoft Research, 2009.

[52] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19:461–471, 1976.

[53] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL 2002: Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[54] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proceedings of the 10th international conference on Model checking software (SPIN 2003)*, pages 235–239. Springer-Verlag, 2003.

[55] Jorg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[56] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[57] Vincent C. Hu, David F. Ferraiolo, and D. Rick Kuhn. Assessment of access control systems. Technical report, National Institute of Standards and Technology (NIST), 2006.

[58] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2nd edition, 2004.

[59] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, April 2002.

[60] Daniel Jackson. Micromodels of Software: Lightweight Modelling and Analysis With Alloy. Technical report, Software Design Group. MIT Lab for Computer Science, 2002.

[61] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.

[62] Trevor Jim. SD3: a trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106 –115, 2001.

[63] James B.D. Joshi, Rafae Bhatti, Elisa Bertino, and Arif Ghafoor. Access-control language for multidomain environments. *IEEE Internet Computing*, 8:40–50, 2004.

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

[65] Masoud Koleini and Mark Ryan. A knowledge-based verification method for dynamic access control policies. Technical report, University of Birmingham, School of Computer Science, Available at: `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/`, 2010.

[66] Masoud Koleini and Mark Ryan. A knowledge-based verification method for dynamic access control policies. In *ICFEM 2011: Proceedings of 13th International Conference on Formal Engineering Methods*, 2011.

[67] Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *ICFEM 2005: Proceedings of 6th International Conference on Formal Engineering Methods*, pages 224–238. Springer-Verlag, 2004.

[68] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6:128–171, 2003.

[69] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL 2003: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 58–73. Springer-Verlag, 2003.

[70] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[71] Jorn Lind-Nielsen. *BuDDy: Binary Decision Diagram Package.* IT-University Copenhagen, Nov. 2002. `http://javabdd.sourceforge.net/`.

[72] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *CAV 2009: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 682–688. Springer-Verlag, 2009.

[73] Alessio Lomuscio and Franco Raimondi. The complexity of model checking concurrent programs against CTLK specifications. In *AAMAS 2006: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 548–550. ACM Press, 2006.

[74] Alessio Lomuscio and Franco Raimondi. MCMAS: A model checker for multi-agent systems. In *TACAS 2006: Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–454. Springer-Verlag, 2006.

[75] Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *AAMAS 2006: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 161–168. ACM Press, 2006.

[76] Radu Mardare and Corrado Priami. Dynamic epistemic spatial logics. Technical report, The Microsoft Research-University of Trento Centre for Computational and Systems Biology, 2006.

[77] Tim Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, 2003.

[78] Prasad Naldurg and Roy H. Campbell. Dynamic access control: preserving safety and trust for network defense operations. In *SACMAT 2003: Proceedings of the eighth ACM symposium on access control models and technologies*, pages 231–237. ACM Press, 2003.

[79] Li Pan, Jorge Lobo, and Seraphin Calo. Extending the CIM-SPL policy language with RBAC for distributed management systems in the WBEM infrastructure. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 145–148. IEEE Press, 2009.

[80] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[81] Prover9-Mace4. `http://www.cs.unm.edu/~mccune/mace4/`.

[82] Hasan Qunoo and Mark Ryan. Specifying and verifying security properties for dynamic access control systems. In *Proceedings of the 24rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, volume 6166 of *Lecture Notes in Computer Science*, pages 295–302, 2010.

[83] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 1999.

[84] Ravi Sandhu. Separation of duties in computerized information systems. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, pages 179–190, 1990.

[85] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2:105–135, 1999.

[86] Ravi S. Sandhu and Venkata Bhamidipati. Role-based administration of user-role assignment: The URA97 model and its oracle implementation. *Journal of Computer Security*, 7, 1999.

[87] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[88] Jan van Eijck and Simona Orzan. Epistemic verification of anonymity. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 168:159–174, February 2007.

[89] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.

[90] Georg Henrik von Wright. Deontic logic. *Mind 60*, pages 1–15, 1951.

[91] Chao Wang, Hyondeuk Kim, and Aarti Gupta. Hybrid CEGAR: combining variable hiding and predicate abstraction. In *ICCAD 2007: Proceedings of the International Conference on Computer Aided Design*, pages 310–317. IEEE Press, 2007.

[92] Daniel S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.

[93] Nan Zhang. *Generating Verified Access Control Policies through Model-Checking.* PhD thesis, School of Computer Science, University of Birmingham, Jun. 2006.

[94] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.

[95] Nan Zhang, Mark D. Ryan, and Dimitar P. Guelev. Evaluating access control policies through model checking. In *ISC 2005: : Eighth Information Security Conference*, volume 3650 of *Lecture Notes in Computer Science*, pages 446–460. Springer-Verlag, 2005.

[96] Conghua Zhou, Bo Sun, and Zhifeng Liu. Abstraction for model checking multi-agent systems. *Frontiers of Computer Science in China*, 5:14–25, 2011.