Policy Invalidation in System Models

Tobias Stig Lindø
s072461

Kongens Lyngby 2012
IMM-PhD-2012-89

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-PhD-2012-89

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

Lyngby, 9-August-2012

Tobias Stig Lindø
s072461

Abstract

We live in a world becoming increasingly complex, and it is equally difficult to ensure safety and security in such systems, especially against insiders. Formalising and analysing these systems can prove beneficial in order to specify or verify any threats or vulnerabilities. In this project, we formalise real-world systems and subsequently perform static analysis to investigate these systems, and check whether they can invalidate any pre-defined system policies. The results show us that policy invalidation, based on static analysis, is useful on the battleground of modern security measures, as it can help to discover vulnerabilities and threats when designing or improving systems. Multiple extensions can yield a more life-like system-modelling tool, increasing both functionality and quality of the threat assessment.

Contents

Preface **i**

Abstract **iii**

1 Introduction **1**
1.1 Project Description 2
1.2 Goals 3
1.3 Report Structure 4

2 Background **7**
2.1 Access Control and Policies 7
2.1.1 Discretionary Access Control 8
2.1.2 Mandatory Access Control 9
2.1.3 Role-Based Access Control 10
2.2 Insider Threats 10
2.3 acKlaim 11
2.4 ANTLR 12
2.4.1 In Practice 13
2.4.2 An Example 13

3 Design **17**
3.1 System Model Design 17
3.1.1 Locations and Edges 18
3.1.2 Actor 19
3.1.3 Action 20
3.1.4 Data 20
3.1.5 Access Policy 20
3.1.6 System Policy 21
3.2 Modelling Language 21

3.3 Policy Invalidation Tool Design 23
3.3.1 Location 23
3.3.2 Actor 24
3.3.3 Action 24
3.3.4 Data 24
3.3.5 Access Policy 24
3.3.6 System Policy 25
3.3.7 Analyser 25
3.3.8 Helper 25
3.3.9 Class Diagram 26
3.4 Pseudo-Algorithm for Analysis 27
3.5 Pseudo-Algorithm for Policy Invalidation 28
3.6 Scenarios 29
3.6.1 Scenario 1 30
3.6.2 Scenario 2 30
3.6.3 Scenario 3 33

4 Implementation **37**
4.1 Tool 37
4.1.1 Main 38
4.1.2 Location 38
4.1.3 Actor 38
4.1.4 Action 39
4.1.5 Data 39
4.1.6 AccessPolicy 40
4.1.7 SystemPolicy 40
4.1.8 Helper 40
4.1.9 Analyser 40
4.2 Modelling Language Grammar 42
4.2.1 Locations 43
4.2.2 Edges 44
4.2.3 Actors 45
4.2.4 Data 45
4.2.5 System Policies 46
4.2.6 Access Policies 47
4.2.7 Basic Rule Set 48

5 Results **49**
5.1 Scenario 1 49
5.2 Scenario 2 50
5.3 Scenario 3 52

6 Discussion 55

6.1 Evaluation . . . 55

6.1.1 Scenarios . . . 55

6.1.2 The Tool . . . 56

6.2 The Tool in Practice . . . 57

6.3 Formalising Problems . . . 58

6.4 Future Work . . . 59

7 Conclusion 61

A Appendix 63

A.1 Using the Tool . . . 63

A.2 Modelling Language Grammar . . . 64

A.3 Scenario 1 . . . 67

Bibliography 69

List of Figures

3.1 Overview of the system 18

3.2 Modelling Language Syntax 22

3.3 Class Diagram for the Tool 26

3.4 Scenario 1 31

3.5 Scenario 2 32

3.6 Scenario 3 34

List of Tables

5.1 Results for the actor *U* scenario 1 50

5.2 Results for the actor *J* scenario 1 51

5.3 Results for the actor *U1* scenario 2 51

5.4 Results for the actor *U2* scenario 2 52

5.5 Results for the actor *U3* in scenario 2 52

5.6 Results for the actor *U* in scenario 3 53

Chapter 1

Introduction

As the structures in the world have grown more complex, be it physical or virtual, the need for controlling this has grown along side it. The airline company needs to ensure the safety of passengers and the robustness of the airplanes, just like a large cooperation needs to safe-guard company secrets or high-value data. But as the airplanes grow more complex, and the buildings grow taller, the methods of control need to grow smarter, as the amount of permutations and possible combinations increase dramatically.

This growth is especially noticed in the fields of Access Control, as the technology available for actors will cover a wider area of technology for each day passed. Thinking about it in practical terms doesn't make it more managable - an office building with dozens of floors, thousands of employees , hundreds of roles, hundreds of thousands of electronic devices... How can one enforce the policies of protecting confidential data in scenes like this? Static analysis.

Static analysis is a method of examining a system without actually running it. An example could be a computer programmer, examining his code without actually executing. But by using static analysis, he can investigate the pattern of his code, ensure that it behaves within the pre-determined boundaries and find possible weaknesses in the structure. Thus, applying static analysis to larger system models, be it buildings or networks, can prove to be very beneficial when searching for possible security breaches, weak links or unintended behavior.

So why is it relevant to search for these weak links in systems? Very often, an intruder will have very limited knowledge of the system, limiting the impact. But one must not forget the existence of *insider attacks*. An insider has numerous advantages over intruders; he is more trusted, he has better access and his knowledge of the system exceeds by far that of an intruder. This opens up a whole new world of possibilities in terms of circumventing the security of a system, and it is thus critical to measure and analyse these possibilities in order to prepare and improve the systems.

You can get very far with the proper access control model, but at the end of the day, you have to place your trust somewhere, and due to insider threats; this choice is crucial and critical for the security of your system. This is where a tool may come in handy; a tool for defining policies to test the validity of your access policies, the amount of trust in your subjects, and the confidentiality of crucial objects.

This inspired the project by yours truly, as this project concerns itself with implementing and evaluating a tool for performing static analysis on system models with the purpose of gaining knowledge about the involved actors and their behavior, and ultimately try to invalidate pre-defined system policies. This allows discovering weak links in system security, or marking actors possessing disputed priviliges.

1.1 Project Description

This project is inspired by the need for analysis in regards to access control, intrusion detection and policy invalidation. The project concerns modelling complete systems, referred to as system models, with locations, actors, data, actions, access policies and system policies, and how this information can be used to invalidate system policies. This type of analysis can be utilized to profile certain scenarios based on the appertaining system policies; is it possible to circumvent the intended behavior and act against the system policies, breaking them? This is the main objective of the tool.

In order to define a language for describing the system models, a process calculus will be utilized. This calculus is called acKlaim, and it is used to model real-world systems, such as organisations, buildings or networks. The modelling language used to describe the system models will be based on the languages used in [CWPN] [CWP08] [CWP]. However, several changes has to be made in order to tailor it for the current tool and appertaining policy invalidation. acKlaim is explained in the Theory chapter, section 2.3 on page 11.

In order to build an interpreter for reading the input system models (also referred to as *scenarios* in this project), we will utilize ANTLR. Since ANTLR yields a framework for creating recognizers, interpreters, compilers and translators, it is well-suited for the needs of this project [ANT]. ANTLR is described thoroughly in the Theory chapter, section 2.4 on page 12.

We will be using ANTLRWorks as the platform for writing ANTLR grammars, as it provides a grammar development environment for ANTLR v3 grammars. ANTLRWorks combines an excellent grammar-aware editor with an interpreter for prototyping, as well as a language-agnostic debugger for isolating grammar errors [Bov].

The underlying structure responsible for maintaining the scenarios in datastructures, as well as the algorithms performing the analysis and policy invalidation, will be programmed in Java on the Eclipse platform. Java was chosen due to its simple syntax and good performance, as well as it rich standard library.

1.2 Goals

Here, the scope of this project will be underlined in terms of goals. These goals are presented in the order of the project flow.

We will investigate and utilize the acKlaim calculus in order to formalise the semantics of our system models. These semantics will later be used to create an interpreter for system models.

In parallel with investigating acKlaim, we will create a number of system models and subsequently analyse them by hand. These system models, and the manually-achieved solutions, will guide the rest of the project, as they state what the analysis should yield.

When the calculus language is determined and the system models are in place, we are ready to build our interpreter for system models in ANTLR. This interpreter is created using ANTLRWorks platform and will follow the already-developed system model semantics.

The next objective is to implement the tool performing the actual analysis and policy invalidation of system models. Using the ANTLR interpreter, the tool will read system models and fill them into appertaining data structures. Once the system model data reside in the tool itself, it will perform the analysis, collecting useful information about actors and their deeds.

The actor results gathered during the analysis aim to describe what locations and data the actor can reach, and under what restrictions. Using this information, the tool will subsequently try to invalidate any defined system policy.

Then, based on the performance of the tool, we will consider the problems and limitations of invalidating policies in system models. Furthermore, we will discuss the benefits of formalising problems in terms of intrusion detection, as well as explain how acKlaim and the developed techniques can be used for modelling and analysing workflows.

1.3 Report Structure

Here follows the report structure.

- The **Abstract** chapter will briefly cover the overall experience and findings of the project.
- The **Introduction** chapter (chapter 1) will cover the basics of the project; the motivation, the overall goals, and finally the report structure.
- The **Backgorund** chapter (chapter 2) will explain the aspects of access control, insider threats, acKLAIM and ANTLR.
- The **Design** chapter (chapter 3) will describe how system models are to be structured, as well as presenting the modelling language used to specify these. This chapter will also describe how the tool was designed, covering both the structure itself and the algorithms for performing analysis and policy invalidation.
- The **Implementation** chapter (chapter 4) will cover how the design was implemented, including practical implementation choices. This includes both the modelling language grammar, as well as the policy invalidation tool.
- The **Results** chapter (chapter 5) will comment on the results gathered from the analysis and policy invalidation. This will include whether the results actually reflected what was expected, or if the tool didn't perform.
- The **Discussion** chapter (chapter 6) will discuss the experience gathered from all phases in the project, as well as cover the results for implementing policy invalidation for acKlaim. Furthermore, the tool itself and the appertaining scenarios are evaluated. This chapter will also explain the usefulness and future of this field of technology.

- The **Conclusion** chapter (chapter 7) will wrap things up; did we achieve the goals for the project?; what did we learn?

Chapter 2

Background

In this chapter, we will cover the theory necessary for understanding the crucial aspects of the project. This includes an introduction to Access Control and Policies, Insider Threats, acKlaim and ANTLR.

2.1 Access Control and Policies

The main purpose of access control is to control who can interact with a certain resource. This type of control is something we see everyday; the lock to your front door, the PIN-code in your local ATM, or your car keys. Access control can be used to secure confidential, or in another way sensitive, information or devices.

In the physical world, access control focuses on who is accessing, where is he trying to go, and when is he going. A undisputed example of this is a simple cipher-lock to an office building. A subject with the correct code may enter, but despite this, the cipher-lock may be programmed to only allow entrance in certain time spans.

In the virtual world of computers, access control covers a much wider field.

Here, authorization, authentication and accountability are vital for ensuring proper access control; authentication states which subject can enter or log-on in a system, authorization determines what the subject are allowed to do, and accountability identifies what actions the subject performed. Furthermore, one should strive to the following approaches in protecting objects [FP]:

Check every access Even though a subject may gain access to an object, this ma not be the case at every enquiry. In some cases, we may want to prevent further access immediately after authorization. Thus, every access should be checked.

Enforce least privilege The *principle of least privilege* states that a subject should possess the smallest amount of objects necessary to perform his tasks. As knowledge is power, allowing subjects to possess more vital information than needed will increase the negative impact of insider threats or feeble-minded accidents.

Verify acceptable usage Allowing access is one thing, but allowing specific actions is another. For example, a subject may be allowed to access a computer, but he may not be allowed to change this system, only read information. Thus, it is equally important to authorize the appertaining actions as to authorize access.

Here follows some of the most recognized access control policies; Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role-Based Access Control (RBAC). Note that both MAC and RBAC are non-discretionary.

2.1.1 Discretionary Access Control

Discretionary access control, known as DAC, is an access policy restricting access to files, or other objects, based on the identity of the subject and/or the groups to which they belong [RG91]. DAC is discretionary in the way that it is applied at your own discretion, i.e. with DAC, it's your choice whether or not to give away your data. For non-discretionary access control models such as MAC; this is not an option. DAC does not only let you define which subject can access your data, it also lets you specify what type of access is allowed. This is an important feature, as you can allow a group of users to read a file but only allow for the group leader to alter it.

In most systems, the following three basic types of access are supported [RG91]:

Read Reads a file.

Write Change or replace a file.

Execute Execute a program file.

2.1.2 Mandatory Access Control

Mandatory Access Control, known as MAC, is an access policy for systems processing sensitive data, where each subject (actors, users, etc.) and objects (files, directories, sockets, etc.) are assigned a sensitivity label. For a subject, this label represents the level of trust associated with this subject, meaning it specifies his clearance. For an object, the label states what level of trust or clearance a subject must own to interact with this object.

In MAC, all access decisions are made by the system [RG91], unlike in DAC, where you, at your own discretion, can specify, who can share your files.

Labeling and MAC as a whole implement a multi-level security policy for handling multiple information classifications at a number of different security levels within a system. Despite the benefits of multi-level security systems, it may yield some frustrations as well. For example, you may experience being able to write a file, yet you can't read it. This is due to MAC's principle of write-up and read-down [RG91], and it is explained in the rules for reading and writing:

Read In order to read an object, the subject's sensitivity level must dominate (equal to or higher) that of the object. This is seen as a read-down. Furthermore, the subject may need to be part of certain categories to access an object. Thus, despite having the appropriate sensitivity level, you need to be in the relevant category as well.

Write In order to write an object, the object's sensitivity level must dominate (again, equal or higher) the subject's sensitivity level. This rule seems odd, since you can actually be too trusted to write to a file. The reason is that one should avoid downgrading information - known as write-down. A subject with maximum clearance could copy contents from an object with maximum sensitivity level to another object with a lesser sensitivity level - this would be a downgrade of the information. In order to avoid this, only upgrades (write-up) of information is allowed.

Furthermore, when using MAC, controlling the imports of information from other systems and export to other system is critical, as every subject and object have a properly maintained sensitivity label, such that sensitive information remains well-protected.

2.1.3 Role-Based Access Control

In larger companies or institutions with many employees, it's complicated to treat each employee individually in terms of access control, as many of them will have the same privileges. Role-Based Access Control (RBAC) addresses this issue, as it lets us associate privileges with groups (aka. roles) [FP]. For example, the group of administrators may have many privileges, whereas the group of janitors might have significantly lesser privileges. Also, if a new janitor joins the crew, all the system needs is to assign this janitor with the privileges of the janitor group, and he is ready to interact with the system.

In RBAC, access in controlled at the system level, as opposed to DAC, where actors are allowed to control access to their own resources. Even though RBAC is non-discretionary, it handles permissions differently than the MAC; MAC controls read and write actions based on the actor's clearance or sensitivity level, while RBAC controls collections of permissions that may include a simple read or write operation, or more complex operations.

Three rules specify the nature of RBAC:

1. A subject can only execute an action, if he has selected or has been selected a role.
2. A subject's role must be authorized. Together with the first rule, this ensures that subjects can only be associated with roles for which they are authorized.
3. A subject can only perform an action if this transaction is authorized for the role of the subject. Together with the first and second rule, this ensures that subjects can only perform actions for which they are authorized.

2.2 Insider Threats

An insider threat is usually hacker (aka. cracker or black hat) with malicious intent. The insider is an employee at some type of cooperation or institution, where he has some form of clearance. What makes an insider dangerous is the trust in him, which is given, as well as his motivation for sabotaging the company from within.

The motivation of an insider come in many forms;

Revenge He may have been fired recently but still has clearance, and now he's looking for revenge.

Financial He is in the need of money, and his trust level in the company can allow him to steal the funds he needs.

Competition He may be working for a competing company and is trying to leak or steal company secrets.

Anarchism He has a desire to perform harmful and destructive actions towards others.

The damage implications of the insider can vary; he can steal secret documents, he can pour water in the servers, or he can inject viruses, worms or trojan horses into office computers. The damage he can cause is directly proportional to the amount of trust, which he possesses, as this yields more ways for him to sabotage the company.

As opposed to an outsider, the insider usually has much better knowledge about the internal system. Although an outsider is suspected to gain some amount of knowledge about the target system, an insider will in most cases be superior in this attribute. This also makes it very hard to protect against insider attacks.

To summarize; insiders have vast information about the target system, their motivation may vary but is always destructive, and the cooperation of the target system trusts them in various degrees. So how can you protect yourself from them? *The principle of least privilege* is beneficial to follow in this case, as it helps reduce the amount of trust given to any employee, insider or not. Said in another way, having a secure internal and robust structure is one of the best defences against insiders. You cannot directly counter-act the insiders, as you do not know their true identity.

2.3 acKlaim

Process algebras are used to formalise systems. By formalising the semantics of systems using process algebras, one can enable the use of tools and techniques for forensic analysis of the systems.

acKlaim is a process algebra, and it is a member of the Klaim family, designed around the tuple space paradigm, where a system contains a set of distributed nodes. These nodes interact through shared tuple spaces by reading and writing tuples. Also inherited from the Klaim family, the acKlaim calculus possesses three layers: nets, processes, and actions. While nets yield the overall structure with processes and tuple spaces, processes perform actions [CWPN].

acKlaim is a variation of the μKlaim calculus, enhanced with access-control primitives and equipped with a reference monitor semantics, ensuring compliance with a system's access-control policy [CWP08]. The reference monitor semantics mentioned ensures that the semantics only performs actions that are specifically allowed, based on the factors; type of action to be performed, identity of the actor performing the action, data in possession of the actor, and any locations involved in the action. The addition of name-annoted processes allows modelling actors moving in systems, which is a major difference to standard Klaim calculi.

To summarize, these enhancements enables acKlaim to be used for static analyses for computing approximations of the consequences of insider attacks [CWPN].

2.4 ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful and flexible language tool with support for many popular programming languages, such as C#, Java, C Python and Ruby. It provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions [ANT]. What is beneficial about ANTLR is that it automates and eases the construction of language recognizers. This is what makes it attractive to use for this project - we need a language recognizer for reading in system models, such that the data can be interpreted for analysis and policy invalidation.

You will need to feed ANTLR with a formal grammar. From this grammar, ANTLR creates a program with the purpose of recognizing input in regards to the target language. Furthermore, by including code in the actual grammar, the recognizer can be evolved to a more sophisticated interpreter.

ANTLR can also provide support for intermediate-form tree construction, tree walking, and error- recovery and reporting.

2.4.1 In Practice

ANTLR will read in your grammar file and generate several source code files, along with other auxiliary files. Most uses of ANTLR will generate either one or both of the following tools [ANT]:

A Lexer reads an input character or byte stream, divides it into tokens using the patterns specified, and generates a token stream as output.

A Parser reads a token stream (as the one generated by the Lexer), and matches the phrases in your language wit the patterns earlier specified.

Under most circumstances, a Lexer and Parser are used together in series to check the word-level and phrase-level structure of the input, hence ensuring that the specified language is followed in the input. Then, they will create an intermediate tree representation, like an Abstract Syntax Tree, and create the final output using a Tree Parser to process the final tree representation. Simpler language tools may skip the intermediate tree and build the actions or output stage directly into the parser [ANT]

2.4.2 An Example

This sub-section will present and discuss a small ANTLR example [ANT]. In listing 2.1 on page 13, you can see the entire grammar file.

Listing 2.1: ex

```
grammar SimpleCalc;

tokens {
    PLUS    = '+' ;
    MINUS   = '-' ;
    MULT    = '*' ;
    DIV = '/' ;
}

@members {
    public static void main(String[] args) throws Exception {
        SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(↩
            args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        SimpleCalcParser parser = new SimpleCalcParser(tokens);

        try {
            parser.expr();
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }
}

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

expr    : term ( ( PLUS | MINUS )  term )* ;

term    : factor ( ( MULT | DIV ) factor )* ;

factor  : NUMBER ;


/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

NUMBER  : (DIGIT)+ ;

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+    { $channel = ↩
    HIDDEN; } ;

fragment DIGIT  : '0'..'9' ;
```

The first line simply states the name of the grammar file. Normally, you would also define the target programming language here, but as the default language is Java, we need not change that.

Below, the `tokens` are being declared. These tokens represent the relation between key-words and their textual representation. Thus, a '+' will be related to the `PLUS`, and so forth.

In the `@members`, a `main` method has been added. This method will create the lexer, the tokens, and the parser. Finally, it will call the `expr` defined in the grammar file.

Below the `main` method, you will find the parser rules. These rules are a recursive way of handling arithmetic expressions. `expr` states that an `expr` may consist of two `terms`, with either an `PLUS` or `MINUS` in between. The '*' in the end states that this can occur any number of times. The `term` defines itself as two `factors`, with either a `MULT` or `DIV` in between. The `term` too can occur any number of times. Finally, `factor` is presented as a `NUMBER`.

In the lexer rules, a `NUMBER` is one or more `DIGITs`. These `DIGITs` are represented by a number between '0' and '9'.

The `WHITESPACE` declares how the grammar will handle whitespaces. Without this, we will receive complaints from the program about spaces, tabs, returns and similar. But it's not enough to simply define the `WHITESPACE`, as the tool will still recognize them in the input. We have to hide it from the parser. This is done by adding the `$channel` flag as `HIDDEN`. The addition of this flag moves the `WHITESPACE`s from the default channel to the hidden channel, letting the parser ignore these occurrences.

Chapter 3

Design

In this chapter, we shall cover the system model design and the modelling language, which later scenarios are to follow. Subsequently, we present the design of the tool, including an UML diagram to show how the tool components shall interact. Then, the design of the algorithms for analysis and policy invalidation is explained, before finally describing the three scenarios that are to be fed into the tool.

Before all this, however, we present a figure of the entire system in order to give you, as the reader, an understanding of how all the different components, files and programs interact. This overview can be viewed in figure 3.1 on page 18.

As figure 3.1 on page 18 shows, the grammar is fed to ANTLRWorks, which returns an *Lexer* and a *Parser*. Using this output from ANTLRWorks, the policy invalidation tool reads in a scenario and performs the analysis and policy invalidation, before finally returning the result.

3.1 System Model Design

This section we introduce the systems, onto which we are applying our model. Here, we touch upon each component; what are their functions and how do they

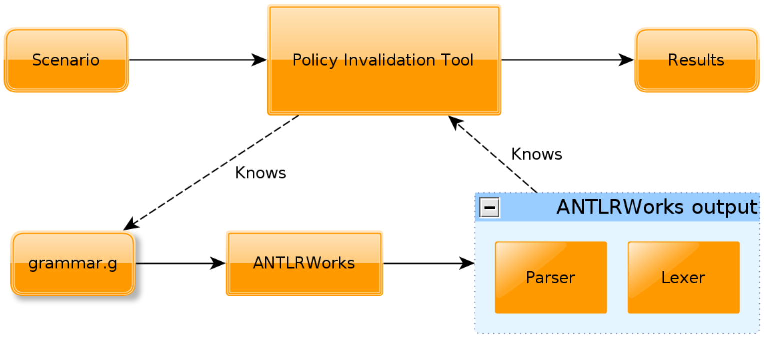

Figure 3.1: Overview of the system

work? This view on system models are inspired by *An extensible analysable system model* [CWP08].

We will now look at each of the following components, and how they are to interact with each other in a system model:

- Locations
- Edges
- Actors
- Actions
- Data
- Access Policy
- System Policy

The design of these components shall guide how they are designed in accordance to the tool. The tool is described in section 3.3 on page 23.

3.1.1 Locations and Edges

The most important components of the system model are the locations. The locations represent the actual rooms in a system model, be it a janitor's closet or a server room. A location can be seen as a node in a graph, where the actor can move to and from. A location contains its own name, a list of access policies and a domain name. The domain name is to be used to classify a collection of locations. For instance, computer locations may belong to the *virtual* domain, while other locations may be a part of the *office*-, *customer*-, or *executive* domain. Apart from rooms, a location can also represent a door, as this may also contain a list of access policies.

Every system model should contain a location labelled *Outside*. This location node will represent the world outside of the system itself, and it simplifies modelling threats coming from the outside.

If a location is seen as a node in a graph, then the edges can be viewed as the directed edges in this graph, meaning they connect locations in a system model. Thus, actors, with the privileges of being able to move to a target location, can move from the original location to the target location, if there exists an edge between the original- and target location.

It's important to note that any movement from a building domain into a virtual domain (fx from the server room to the actual server) is one-way, meaning there should be no returning edges. The reason is that the system should only allow for actors to enter the virtual domain, and move around in the shape of a process, but it should not allow for this process to become part of the "physical" realm again. For instance, if an actor interacts with PC1 in the virtual domain, there's no harm in allowing him to return to the room from where he came, but if he moves through PC1 to PC2 as a process, he should not be allowed to move from PC2 to its originating location, as this would allow actors to actually move through wires and circumvent access policies.

3.1.2 Actor

An actor is a core component of the system model. Its goal is to simulate the movement and behavior of certain individuals or roles. An actor should possess a name, as well as a starting location. From this starting location, the actor will move throughout the system model, visiting all possible locations and performing all possible actions. These actions may differ in reality, but in a system model, it may for example involve picking up certain data. For the purpose of analysis, an actor should also keep track of what locations he visits, what data he picks up, or what restrictions he passes through access policies.

3.1.3 Action

An action incarnates an entity that an actor can perform to change his own state or a state of an object near him. An action may be to pick up (read) a document in an office, evaluate a computer system, or move to a adjacent location. The collection of available actions is as follows:

- Input
- Output
- Evaluate
- Read
- Move

The Move action will allow an actor to move from this location to appertaining exit locations, if the actor can pass their access policies. Locations in the virtual domain may also yield the Move action for actors, allowing them to move from one virtual location to another as a process.

The Read action lets an actor read a piece of data, in case the data contains one or more access policies.

3.1.4 Data

A data object is seen as a container for information. This information can differ in its form, be it an actual document of company secrets, a physical key used to unlock the janitor's, or a little note with the access code of a careless employee on it. Data can reside at a location or with an actor.

But data are not necessarily available for an actor, just because the actor can reach it. Thus, data may own a list of access policies, which define what actors are able to perform which actions on this piece of data.

3.1.5 Access Policy

An access policy is a rule. It states an accessor and appertaining actions, which the accessor are allowed to perform. An access policy can reside with any kind of object in the need of access control, be it a location or data.

For example, an access policy may state that only the administrator is allowed to venture on, or only someone in possession of the janitor's key can pass, or only the user 'Carl' is allowed to read this document. Furthermore, if data is situated in a safe, its access policies can state that only actors possessing the data object that is the right combination can interact with it.

3.1.6 System Policy

Like with the access policies, a system policy can be seen as a rule, yet its implications differ dramatically. While the purpose of an access policy was to apply access control, the purpose of a system policy is to apply after-the-math policy invalidation. Said in another way, **system policies are the type of policies that this tool aims to invalidate** to yield useful information about insider threats.

A system policy is simple in nature; it contains the name of the accesser and the name of the placement. Combining this results in a policy stating:

$$\textit{OBJECT must } \underline{\textit{not}} \textit{ be at PLACEMENT}$$

The object can point to either data or an actor, while the placement may describe either a location or an actor. Despite this, not all combinations should be allowed - this is explained in section 3.2 on page 21.

3.2 Modelling Language

This section cover the Modelling Language used. As noted in the Introduction chapter, 1 on page 1, this language is inspired from previous intrusion detection projects [CWP08] [CWPN], and it is based on acKlaim. Its syntax is based on the actual system model design, which is described in section 3.1 on page 17. The syntax of this language can be seen in figure 3.2 on page 22.

The system model, or scenario, consists of locations, edges, actors, data and system policies. A location is represented by a location name, a list of access policies of that location, and a domain name. An access policy contains an accesser name and a list of appertaining actions. An action can be any of the four following actions;

$$
\begin{array}{rcl}
Scenario & ::= & locations : \{Locations\} \\
 & & edges : \{Edges\} \\
 & & actors : \{Actors\} \\
 & & data : \{Data\} \\
 & & systemPolicies : \{SystemPolicies\} \\
Locations & ::= & Location* \\
Location & ::= & LocationName\{AccessPolicies\}(DomainName); \\
AccessPolicies & ::= & AccessPolicy* \\
AccessPolicy & ::= & AccesserName : Actions; \\
Actions & ::= & Action* \\
Action & ::= & i \\
 & & |o \\
 & & |e \\
 & & |r \\
 & & |m \\
Edges & ::= & Edge* \\
Edge & ::= & LocationName- > LocationNames; \\
Actors & ::= & Actor* \\
Actor & ::= & ActorName@StartLocationName; \\
Data & ::= & DataItem* \\
DataItem & ::= & DataName\{DataPolicies\}?@ActorName; \\
 & & |DataName\{DataPolicies\}?@LocationName; \\
DataPolicies & ::= & AccessPolicy* \\
SystemPolicies & ::= & SystemPolicy* \\
SystemPolicy & ::= & DataName!@ActorName; \\
 & & |DataName!@LocationName; \\
 & & |ActorName!@LocationName;
\end{array}
$$

Figure 3.2: Modelling Language Syntax

i Input

o Output

e Evaluate

r Read

m Move

An edge contains a location name and a comma-seperated list of location names, defining all the exits of this location, while an Actor contains an actor name and the name of the starting location. A Data item owns a name, possibly a list of data policies, and the name of its placement (an actor or a location). If data does not contain any data policies, it means that anyone can manipulate with the data item.

The basic form of a system policy is an object name and a placement name. In reality, this can construct three valid combinations:

- A data name and an actor name
- A data name and a location name
- An actor name and a location name

3.3 Policy Invalidation Tool Design

Here, we explain how the tool itself was designed to match the system model design, covered in section 3.1 on page 17. We will also present a class diagram of the tool as a showcase of the different classes to be implemented, and how they should interact.

3.3.1 Location

A location should, beside its name and domain name, contain a list of exit locations. When an actor visits a location, he will thus be able to view its exit locations and investigate whether he can move to any of them. An actor should also be aware of any access policies at this location, as this will allow him to check his rights at this location; am I allowed to move from this location, or am I allowed to read information at this location? Finally, a location should present an actor of what data are available.

3.3.2 Actor

Basic attributes of an actor include his name and his starting location. But an actor also needs to keep track of various analysis results. His owned data contains the data in possession when entering the system mode, while known data consists of data discovered during the scenario. A list of locations will incarnate what locations the actor can reach in the scenario, and the actor will also keep track of the restrictions of reaching his reachable locations and the restrictions for reaching and picking up known data.

3.3.3 Action

An action instantiates what an actor can perform. The type of action should vary from a pre-defined set of available actions, as the ones mentioned in section 3.1.3 on page 20.

3.3.4 Data

Every data component should have a name and a list of access policies. These access policies define the restrictions of interacting with the data. An actor should be able to ask a piece of data, if he is allowed to read it; if his name matches an accessor in a access policy, or if he possesses the necessary data (i.e. a key or code). If a data component doesn't contain any access policies, everybody can interact with it.

3.3.5 Access Policy

An access policy should follow a very simple design while offering the necessary functionality, as stated in its system model design, section 3.1.5 on page 20. An access policy contain an accessor, pointing either to an actor name or a specific data item, allowing access. The accessor name is paired up with a list of allowed actions, should the accessor be confirmed. An actor should be able to ask an access policy whether everybody can pass, i.e. the accessor matches the *ANY* actor.

3.3.6 System Policy

The system policy in the tool closely follow the design of a system policy in system models, section 3.1.6 on page 21. It should have an object name and a placement name, where the object name can refer to an actor or data, and the placement name can refer to a location or an actor.

In the tool, system policies are per nature *negative*, meaning when a system policy is defined with an object name and placement name, it will always state that the object *must not* be at the placement. The reason is that it reduces the complexity of system policies greatly, and extending the system policy design with a true/false flag may still be possible.

3.3.7 Analyser

The main purpose for the analyser is to contain the algorithms for performing system model analysis and policy invalidation. Thee algorithms are covered below in section 3.4 on page 27.

Aside from containing these algorithms, the analyser shall have the actual system model in the tool's data structures; a collection of locations, a collection of actors and a collection of system policies. These data structures yield all the information needed to perform the system model analysis and policy invalidation.

3.3.8 Helper

The purpose of the helper is to, as the name implies, assist the analyser. The helper will contain pre-defined expressions, determining how to recognise the *ANY* actor. It will also have the key-words for identifying the available domain names, in regards to locations.

The helper can easily be extended with further functionality or key-word information, in parallel with the tool itself being improved or extended.

3.3.9 Class Diagram

In this subsection, we present a class diagram of the tool's structure. The purpose is to show which classes utilizes each other. The class diagram can be seen in figure 3.3 on page 26.

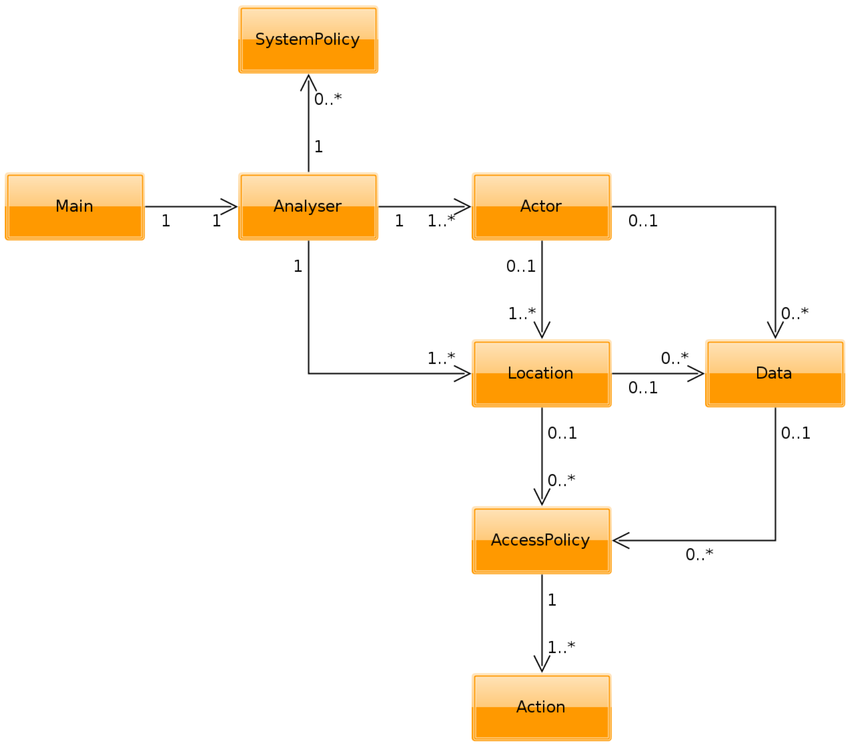

Figure 3.3: Class Diagram for the Tool

The class diagram shown in figure 3.3 on page 26 presents the associations between different classes in the tool. The numbers situated on the edges explain the quantitative relationship between the source- and target class.

The `Analyser` can contain zero or more `SystemPolicys`, while one or more `Locations` or `Actors` are needed for a proper scenario. An `Actor` should be associated with one or more `Locations`, but not necessarily any data. A `Location` may contain any amount of `Data` and `AcccessPolicys`, while an `AccessPolicy` should always possess on or more `Actions`.

3.4 Pseudo-Algorithm for Analysis

This section contains the pseudo algorithm for the analysis and the appertaining thoughts to its nature, behavior and complexity. The pseudo-algorithm can be viewed in listing 3.1 on page 27.

Listing 3.1: Pseudo-algorithm for performing system model analysis

```
Stack<Location> stack;

Restrictions currenRestrictions;

Actor currentActor;
Location currentLoc;

for each actor a
    Set currentActor to a;
    Push the starting location of currentActor to stack;
    Add the starting location of currentActor to his list of reachable↩
         locations

    while(stack is not empty)
    Set currentLoc to the latest entry in the stack (pop it)
    Set currentRestrictions to that of currentLoc;

    if(currentLoc has any restrictions)
        Add restrictions to currentRestrictions;

    if(currentLoc has any data)
        if(currentActor has permission to interact with the data)
        Set restriction of data to currentRestrictions;
        currentActor picks up data;

    if(currentActor has gained new knowledge)
        Push all his reachable locations to the stack;

    if(currentActor has permission to move from currentLoc)
        for each exit location in currentLoc
        if(currentActor can reach exit location)
            Set restrictions of exit location to currentRestrictions;
            Push exit location to stack;
            Add exit location to reachable locations of currentActor;
```

The algorithm iterates through each actor. For each actor, set him to currentActor and add his starting location to the stack, as well to his list of reachable locations. From here on, it continues to pop locations from the stack and onto the currentLoc, while the stack is not empty. For each iteration in the while-loop, it:

- Sets currentLoc to the popped location
- Sets currentRestrictions to those of currentLoc

- If currentLoc has any restrictions add these to currentRestrictions
- If currentLoc has any data, try to pick them up. If this succeeds, data should inherit the currentRestrictions.
- If currentActor has gained new knowledge, push all his reachable locations to the stack
- If currentActor has permission to move, iterate through each exit of currentLoc. If the currentActor can move to any of these, push it to the stack, as well as adding it to the reachable locations of currentActor.

The way this pseudo-algorithm moves is quite beneficial for simplicity. Every location is only visited once by the actor, unless the actor has learned new knowledge (i.e. data). The reason for pushing all reachable locations to the stack again, if the actor gains new knowledge, is to allow him to try and put this new knowledge into usage. The actor might have got a hold on the key, which he can use to open up the server room. Thus, by re-pushing all reachable locations to the stack in this case, the order of which rooms the actor visits or which data he picks up is not crucial, as he will eventually visit all locations again, if he picks up new data.

The pseudo-algorithm in 3.1 on page 27 also confirms what kind of data, we are gathering as results. The actor will possess a list of reachable locations. The restrictions added these location tells what data are needed for the actor to reach and enter this location. The restrictions of data states what data are needed to reach and pick up this data. All these recorded results are actor-specific, as actors shall have individual data and thus rights. The user may reach the server room using different data than the janitor.

3.5 Pseudo-Algorithm for Policy Invalidation

This section presents the pseudo algorithm for the policy invalidation. This section shall also house the thoughts behind the design of this algorithm. The pseudo-algorithm can be viewed in listing 3.2 on page 28.

Listing 3.2: Pseudo-algorithm for performing policy invalidation

```
for each system policy sysPol
    if the sysPol.ObjectName is an Actor a
    //By policy rules, the PlacementName is then a location
    if a.reachableLocations contains PlacementName
        POLICY INVALIDATED;
```

```
else //By policy rules, ObjectName matches Data d
if PlacementName matches Actor a
    if a.knowsData(d)
    POLICY INVALIDATED;

else if PlacementName matches Location l
    for each actor a
    if a.knowsData(d) && a.hasReachable(l)
        POLICY INVALIDATED;
```

The pseudo-algorithm iterates through each system policy in the system model. Then, using if-conditions, it checks the nature of the system policy and looks for any breaches of the system policy:

- If the object name matches an actor
  - By policy rules, the placement will match a location
  - If the actor can reach the location, the policy is invalidated
- Else, by policy rules, the object name matches a data name
  - If the placement name matches an actor
    - If the actor knows the data, the policy is invalidated
  - Else if the placement name matches a location
    - For each actor
      - If the actor knows the data and can reach the location, the policy is invalidated

Due to the design of actors, it is trivial to look up his list of reachable locations. Thus, if the object name is an actor, the algorithm looks through the actors reachable locations for the location with the placement name. Likewise, if the accessor does not match an actor, policy rules dictate that the object is a data component. From there on, it's easy to check whether the placement actor knows this data, or if any actor both knows the data and can reach the placement location.

3.6 Scenarios

Based on the model design (section 3.1 on page 17) and modelling language (section 3.2 on page 21), we will now present three different system models, also referred to as scenarios. These scenarios help define what we expect from the implemented solution. Each of the scenarios represent a distinct situation of access control conflicts, which the implemented solution should be able to handle. While we discuss and analyse each scenario here, the real scenario text files can be viewed in the appendix, section A.3 on page 67.

In the scenarios, access policies are in black boxes, data objects are in blue boxes, and system policies are in read boxes.

3.6.1 Scenario 1

The first scenario is adopted from *An extensible analysable system model* [CWP08]. This was utilized, since it yields a case with a broad scope of both opportunities and several kinds of actors. This scenario was altered due to the scope of the tool, and it doesn't support logging or locations with special actions available (printers or similar). Scenario 1 can be viewed in figure 3.4 on page 31.

In scenario 1, a user *U* and a janitor *J* starts in the *Out* node. While *U* possesses a the code *code_ U*, the janitor has both *code_ J* and *key_ J*. The user can, using his *code_ U*, reach both *Svr* and *Usr*, while the janitor, using his *code_ J* and *key_ J*, can reach both *Svr* and *Jan*. However, the user can also interact with *PC* 1 and 2, and even pick up the *secret_file* in *PC2*.

The system policies in this scenario simulate a company with concern of the *secret_file*, i.e. it must not exit the building or fall into the wrong hands. Thus, two system policies are defined; one states that the *secret_file* must not reach *Out*, and the other states that the janitor must not be in possession of the *secret_file*. While the janitor cannot get a hold on the *secret_file*, the user can, and he is also able to carry it to the *Out* node, breaking the system policy! But "oh no" is not the correct response to this, as the purpose of this tool is to actually invalidate policies and yield beneficial information about the restrictions appended to the actors, as well as any data. To summarize, the user will break a policy by carrying the *secret_file* to the *Out* node, but it only proves him to be a trusted, or in some perspective the weakest, link, as the integrity of the *secret_file* stands or falls with his reliability.

3.6.2 Scenario 2

The second scenario was created with the motive of showing how the tool behaves when access policies in the virtual domain are crucial for invalidating a policy or not. Scenario 2 can be seen in figure 3.5 on page 32.

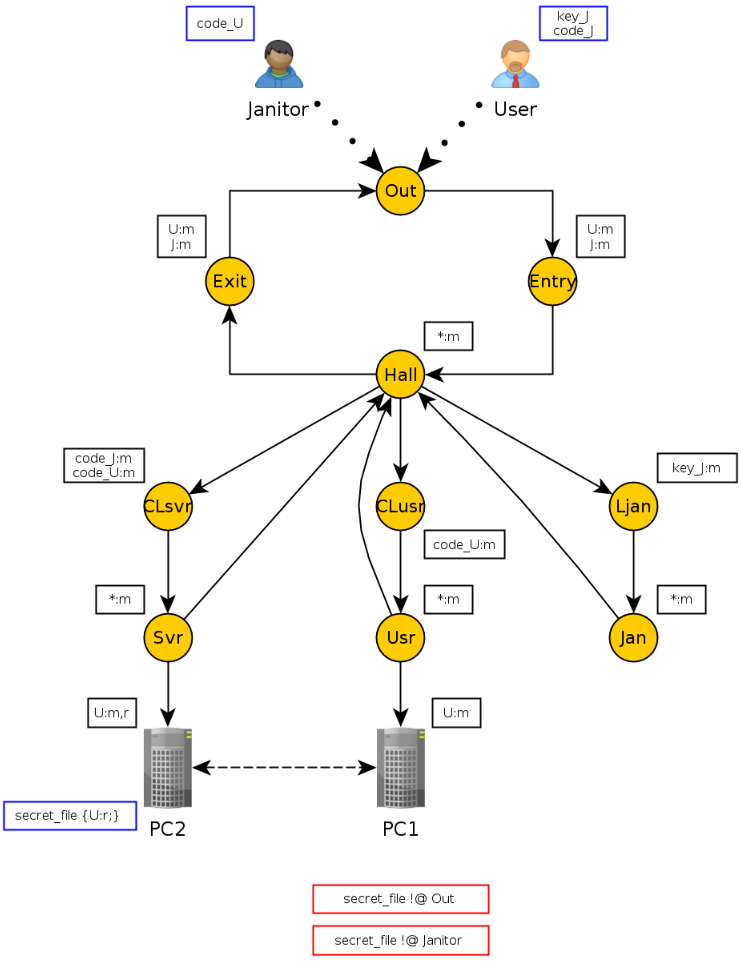

Figure 3.4: Scenario 1

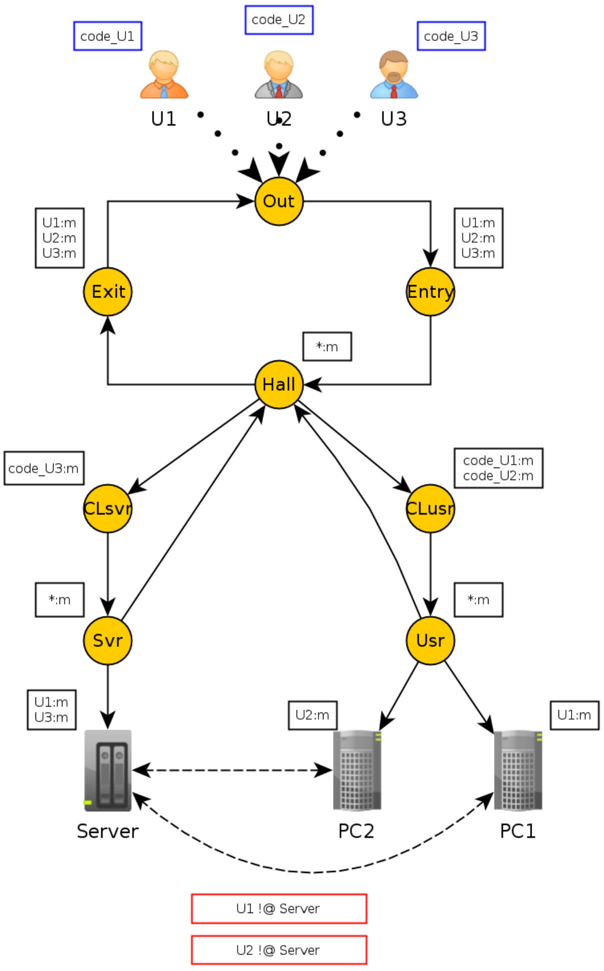

Figure 3.5: Scenario 2

The thought behind scenario 2 is that while the actor *U3* may have direct access to the server, other users may be able to access it through virtual locations, bypassing the server room's access control, and thus relying solely on the access control of the server itself. The actors *U1* and *U2* have each their own PC, which is linked to the *Server*. Yet, only *U1* is allowed to access the server of the two.

This scenario simulates how a company may have changed the physical access policies but have forgotten to alter the virtual access policies as well, allowing an actor to access the *Server*. In reality, this could've been a demotion of an administrator, who then could access the server and sabotage it as an act of revenge. The system policies only look for actors beside the administrator (*U3*), to access the server. This scenario will invalidate the policy stating that *U1* must not be at the *Server*. Although *U2*'s PC is linked with the *Server*, he is missing the necessary rights to access it and can thus be ruled out as a threat.

3.6.3 Scenario 3

The third scenario was created to show of a more complex twist; namely how an actor can sequentially gain more and more new knowledge, which finally allows him to break a system policy. Scenario 3 can be viewed in figure 3.6 on page 34.

Scenario 3 offers a case with only one actor; *U*, yet several branches demanding for different codes. The motivation behind this scenario is to show how the tool can handle how the state of an actor changes, when he picks up new data. The system policy in this scenario states that the actor *U* must not reach the storage room *Sto*. That system policy can be broken, but only if *U* picks up both the *code_B* and the *code_C* key, as both of these are needed to reach *Sto*. Because of this, scenario 3 also yields a perfect example of how the tool can handle data restrictions, as the flow of breaking the system policy will be along the lines of:

- *U* uses facial-recognition to enter the hall.
- *U* uses *code_A* to enter *USR* and picks up *code_B*.
- *U* uses *code_B* to enter *Office* and picks up *code_C*.
- *U* enters *Svr* using *code_B*.
- *U* moves to *Sto* using *code_C* -> system policy invalidated.

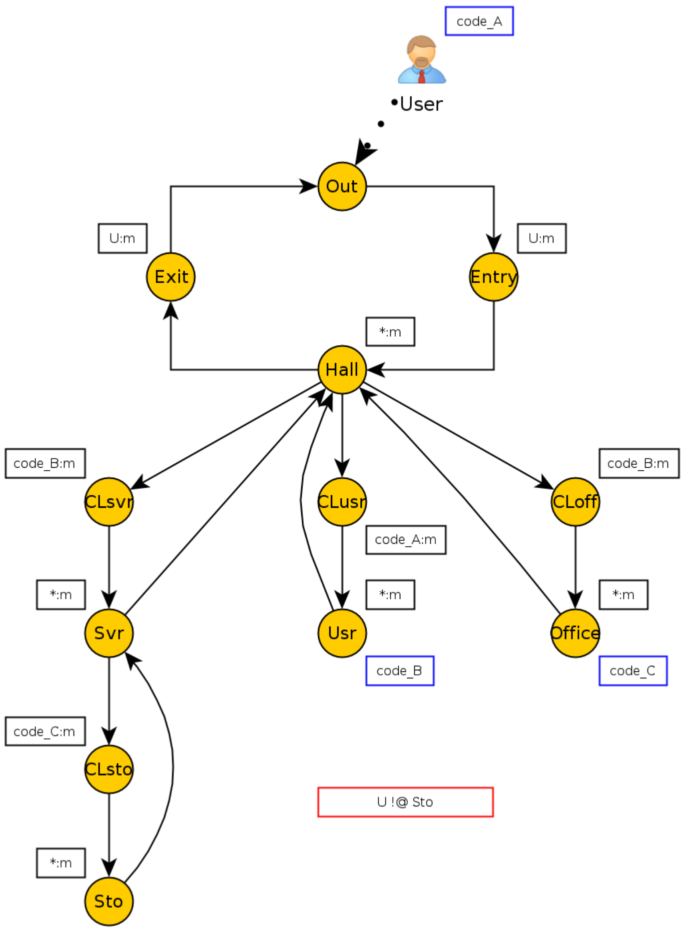

Figure 3.6: Scenario 3

This scenario presents a case, where a company may be relying on spreading out crucial information in order to increase security of the storage facility. Despite this tactic, the tool can point out how an actor can gain access to the necessary keys and enter the forbidden storage room.

Chapter 4

Implementation

This chapter contains the implementation. Here, we will cover the implementation of the tool for analysis and policy invalidation. Subsequently, the implementation of the grammar for creating a system model interpreter using ANTLR will be explained.

Please note that in this chapter and onwards, any methods or classes from the tool are written with `typewriter`, while any method or entity from the modelling language grammar is written with *italic*.

4.1 Tool

In this section, we will explain how the tool was implemented in Java. This section will not contain any code snippets, but investigating the code in parallel may yield a better understanding, if need be. Each class will be covered, including note-worthy methods.

4.1.1 Main

In the `Main` class, we use the *Lexer* and *Parser* generated by ANTLRWorks to read in the input scenario and create the `Analyser`. Once the `Analyser` has been created, we call its public methods:

`analyze` performs the scenario analysis.

`policyInvalidation` checks whether any `SystemPolicy` has been invalidated.

`printActorResults` prints the results gathered for each `Actor`.

If getting the `Analyser`, or any of the above-mentioned public methods, fails, we print the caught exception.

4.1.2 Location

A location keeps track of its name and domain by two `String` fields. It has an `ArrayList<Location>` to contain any exit locations, to which an visiting `Actor` may try to move to. It also possesses its own `HashSet<AccessPolicy>`, which is used by the `Location` when an `Actor` asks for permission to move to here. Finally, an `ArrayList<Data>` incarnates the `Location`'s available data.

One public method, `boolean actorCanAccess`, checks whether an input `Actor` should be allowed to reach this `Location`. For starters, if the `Location` has no `AccessPolicys`, it simply allows him to pass. If any `AccessPolicys` are present, it iterates through them, checking for any case of allowance.

Another public method, `AccessPolicy getActorAccess`, fetches the `AccessPolicy`, which allows an input `Actor` to pass. If more than one `AccessPolicy` may allow access, the first one encountered by the iterator is returned.

These methods will allow an `Actor` to investigate, and if possible, get the `AccessPolicy`, which allows him to access. This `AccessPolicy` is needed by the `Actor`, since it will be added to his restriction results.

4.1.3 Actor

An `Actor` contain numerous associations with other classes, as it not only must maintain the needed information to iterate through a scenario, but it should also keep track of its own results.

The `Actor` possesses a name and the name of his starting location, both represented by a `String`. While the `ArrayList<Data> knownData` contains the `Data` learned during the scenario iteration, the `ArrayList<Data> ownedData` contains the `Data`, which the `Actor` starts with. An `ArrayList<Location>` keeps track of any visited `Location`. This is later shown as a result, but it may also be used by the tool, if the `Actor` has learned new knowledge and needs to back-track to previously-visited `Locations`. Finally, two `HashMap<String, ArrayList<String»` incarnate the `Data` restrictions and the `Location` restrictions, which tell what restrictions the `Actor` has passed to reach a `Location` or reach and interact with `Data`.

The public method `void setLocationRestrictions(String locationName, ArrayList<String> restrictions)` is an important method, as it will add the `Actor`'s current restrictions to the appertaining current `Location` in `locationRestrictions`. Together with the `reachableLocations`, it will ultimately yield a list over all the `Locations` reachable, as well as the constraints of getting there.

Another public method, `pickUpData(Data data, ArrayList<String> restrictions)`, is used by the `Actor` to pick up new `Data`. It adds the `Data` to `knownData`, and sets its corresponding restriction in the proper hash map: `dataRestrictions`.

4.1.4 Action

The simple `Action` class contains a single `action`, which may be any of the defined `enum Actions {READ, EVAL, IN, OUT, MOVE}`. For each of these, it possesses a `static` constructor.

4.1.5 Data

The `Data` class has a name, represented by a `String`, and a `HashSet<AccessPolicy>` to represents its access policies.

It contains the public method `boolean actorCanReadData(String actorName)`, which is used by `Actors` in the `Analyser` to ask whether he can interact with this `Data` component or not. If the method has no `AccessPolicys`, he is allowed access right away. If it does, a `while` loop will iterate through each `AccessPolicy`, until one letting the `Actor` pass is met. If none matches the criteria, the `Actor` will be denied interaction with the `Data`.

4.1.6 AccessPolicy

An `AccessPolicy` uses a `String` to represent the allowed accesser name, and a `HashSet<Action>` as the set of `Action`s.

The public method `boolean allCanAccess` checks whether the accesser name equals the `anyActor`, and that the set of actions contains the `Move Action`. If both these hold true, all `Actor`s are allowed to access.

The `allCanAccess` method utilizes another public method; `containsMoveAction`. This method is used to check for the occurrence of the `Move Action` in the set of actions, and it is used not only by the `allCanAccess` method, but also by the `Analyser`, where it functions as a flag for whether an `Actor` can move from a `Location` or not.

4.1.7 SystemPolicy

The `SystemPolicy` class only contains two `String` fields; one for the object name and one for the placement name. These fields are used to identify one of the valid combinations presented in the design chapter, section 3.2 on page 21.

Having these fields as `String`s makes it painless to identify the nature of the `SystemPolicy`, as one only needs to match the object- and placement names with the corresponding `Actor`-, `Data`-, or `Location` names.

4.1.8 Helper

The `Helper` class assists the `Analyser` by containing and offering methods for returning three `String` identifiers. These identifiers include the `anyActor`, the `building` and the `virtual`. While the first is used in an `AccessPolicy`, when all `Actor`s are allowed to pass, the latter two denote possible domain names for a `Location`.

4.1.9 Analyser

The `Analyser` is where it all goes down, as this contains all the data structures representing the input scenario, as well as the algorithms for performing the analysis and policy invalidation. The behavior of these algorithms are explained in the design chapter, section 3.4 on page 27.

The `Analyser` has the following fields; a `HashMap<String, Location> locations` for the scenario locations, an `ArrayList<Actor> actors` for the active actors, and an `ArrayList<SystemPolicy> systemPolicies` for the system policies to be trialled.

The public `analyze` method performs the system model analysis. It iterates through the `actors` and performs the appertaining analysis. During this analysis, the `locations` hash map is used for fetching any `Location` given its `String` name, as this is the *key* of the hash map. A `Stack<Location> stack` is used to keep track of the `Locations` to visit. Thus, whenever an `Actor` encounters a not-before-visited `Location`, which he is allowed to reach, it will be added to the `stack`. When reaching a `Location`, the `Actor` will inherit the restrictions of that `Location`. Whenever the `Actor` finds a `Location`, which he can reach, it is added to the `stack` and inherits the `Actor`'s current restrictions. As the current restrictions are based on the `AccessPolicys` passed, at the end of the analysis each `Location` will possess the restrictions for the current `Actor` to reach it. Likewise with `Data`; these objects inherit the current restrictions of an `Actor` when picked up, and will ultimately yield the restrictions necessary for reaching and picking the `Data` up. These restrictions, together with the reachable locations, are the results gathered from the `analyze` method.

The public method `policyInvalidation` iterates through each `SystemPolicy`, using a `for`-loop, and uses `if` conditions to identify the nature of the `SystemPolicy`. It looks up the object name and placement name in the `locations`- and `actors` datastructures to find a match, and when identified, tries to invalidate the policy:

- If a `Actor !@ Location` is encountered, it loops through the reachable `Locations` of the `Actor`. If the corresponding `Location` is met, the `SystemPolicy` has been invalidated.

- If a `Data !@ Actor` is met, it iterates trying to find the `Actor` with the matching name, who also knows the `Data`. If this is fulfilled, the `SystemPolicy` has been invalidated.

- If a `Data !@ Location` is met, the algorithm looks for `Actors` in possession of the `Data`. If any of these `Actors` can also reach the policy `Location`, the `SystemPolicy` has been invalidated.

4.2 Modelling Language Grammar

In this section, we will cover how the grammar file, used by ANTLR to create a system model interpreter, was implemented. We will approach this by segmenting the grammar into several bits and explain them individually. The un-segmented grammar can be found in the appendix chapter, section A.2 on page 64. The grammar file follows the language specified in the design chapter, section 3.2 on page 22.

Listing 4.1: Grammar file setuup

```
grammar PolicyInval;

@header {
package output;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import dom.*;
}

@lexer::header
{
    package output;
}

@members
{
    private Analyser analyser;
}
```

Listing 4.1 on page 42 shows the setup in the grammar file. Here, the name of the grammar file itself is declared. Furthermore, the package name, imported java data structures, and the tool's data structure itself is imported. Finally, the `Analyser` is declared as a private member.

Listing 4.2: Grammar file Analyser setup

```
getAnalyser returns [Analyser an] :
    {
        HashMap<String, Location> locations = new HashMap<String,
Location>();

        List<Actor> actors = new ArrayList<Actor>();

        List<SystemPolicy> systemPolicies = new
ArrayList<SystemPolicy>();
    }

  //---------------------------------------
```

```
//Intermediate methods are placed here
//------------------------------------
{
  //Creates a new specification analyser with input locations , ↩
      actors and
  //system policies
  an = new Analyser(locations , actors , systemPolicies);
}
;
```

In listing 4.2 on page 42, the grammar method for retrieving the `Analyser` is shown; *getAnalyser returns [Analyser an]*. Initially, it declares the data structures needed to create an `Analyser`, namely a `HashMap<String, Location>` for *locations*, a `List<Actor>` for *actors*, and a `List<SystemPolicy>` for *systemPolicies*. Hereafter follows a series of operations to retrieve the information from the input system model file. These operations are explained individually later on. When all the system model data has been read into the data structures, the `Analyser` is created. As the attribute of the `getAnalyser` method states, it returns the `Analyser` after creating it.

Now follows the explanations of methods and supportive operations for retrieving locations, edges, actors, data and system policies.

4.2.1 Locations

In the *getAnalyser* method, the operations in listing 4.3 on page 43 are used to retrieve a `Location` and place it in the appertaining data structure. In this code-snippet, we define a repeated process inside of the '{' and '}' brackets, exceeding *locations*. For each process, we enter the *location* operation seen in listing 4.4 on page 44. This is the actual code interpreting a line of an location. It divides every single line according to the modelling language, creates a new location, and returns it to the caller, being the *getAnalyser* method.

Listing 4.3: Retrieve a location and place in data structure

```
'locations'
    '{'
        (
            l = location
            {
                locations.put(l.getName(), l);
            }
            ';'
        )*
    '}'
```

Listing 4.4: Create a location from line contents

```
location returns [Location l] : // a location
    LOCNAME = NAME
    ('{'
        policies = accessPolicies
    '}')
    '('
        DOMAINNAME = NAME
    ')'
    {
      // A location is created with ID, set of attributes and set of ↩
          policies
      l = new Location ($LOCNAME.text, policies, $DOMAINNAME.text);
    }
  ;
```

4.2.2 Edges

For interpreting edges, the *getAnalyser* first divides the line by the occurrence of '->'; the name before this is the target location, and the list of names after that yield its exit locations. For each exit location, it fetches the actual location from the data structure and adds it to the target location's list of exit locations. To manipulate with the exit locations, they are processed by the *exitLocations*. This operation, viewable in listing 4.6 on page 44, interprets the exit locations as a comma-seperated collection and adds each exit location to a list. This list is returned to level of *getAnalyser*.

Listing 4.5: Add exits to locations

```
'edges'
    '{'
        (
            CURLOC = NAME
            '->'
            exits = exitLocations
            //Add exits to the CURLOC
            {
                Location tempLoc;
                for(int i = 0; i < exits.size();i++)
                {
                    tempLoc = locations.get(exits.get(i));
                    locations.get($CURLOC.text).addExit(tempLoc);
                }
            }
            ';'
        )*
    '}'
```

Listing 4.6: Create and return list of exit locations

```
exitLocations returns [ArrayList<String> ex] :
    {ex = new ArrayList<String>();}
    firstLoc = NAME
    {ex.add($firstLoc.text);}
    (
        ','
        otherLoc = NAME
        {ex.add($otherLoc.text);}
    )*
    ;
```

4.2.3 Actors

In listing 4.7 on page 45, it is presented how actors are read from file. For each occurrence, it defines an `Actor` with a name and a starting location, and adds him to the appertaining data structure.

Listing 4.7: Add actor to list of actors

```
'actors'
    '{'
        {Actor a;}
        (
            ACTORNAME = NAME
            '@'
            STARTLOC = NAME

            {
                a = new Actor($ACTORNAME.text, $STARTLOC.text);
                actors.add(a);
            }
            ';'
        )*
    '}'
```

4.2.4 Data

Here we explain how data are interpreted from the input file. In listing 4.8 on page 45, it shows how data are processed by first identifying its name, and then where it should reside. Since data can be placed at either a location or an actor, we have to check for the occurrence of the *DATALOCATION* name in either of these data structures. If we meet a match, we append the data to the appropriate component.

Listing 4.8: Create data from line contents and place accordingly

```
'data'
    '{'
        {Data tempData;}
        (
            DATANAME = NAME
            (
                '{'
                    dataPolicies = accessPolicies
                '}'
            )?
            {tempData = new Data($DATANAME.text, dataPolicies);}
            '@'
            DATALOCATION = NAME
            {
                //If the Data is placed at a Location
                if(locations.containsKey($DATALOCATION.text))
                {
locations.get($DATALOCATION.text).addData(tempData);
                }
                else //Else, the data is placed at an Actor
                {
                    for(int i = 0; i < actors.size();i++)
                    {

if(actors.get(i).getName().equals($DATALOCATION.text))
                        {

actors.get(i).addOwnedData(tempData);
                        }
                    }
                }
            }
            ';'
        )*
    '}'
```

4.2.5 System Policies

In listing 4.9 on page 46, we see how a system policy is recognized as a *systemPolicy* and added to the appropriate data structure. How the system policy is actually interpreted from line contents is shown in listing 4.10 on page 47, where each line is separated by '!@', and the object name and placement name is identified and used to create and return a system policy.

Listing 4.9: Add system policy to data structure

```
'systemPolicies'
    '{'
        (
            sysPol = systemPolicy
            {
                systemPolicies.add(sysPol);
            }
            ';'
        )*
    '}'
```

Listing 4.10: Create a system policy from line contents

```
systemPolicy returns [SystemPolicy sys] :
    OBJECTNAME = NAME
    '!@'
    PLACEMENTNAME = NAME
    {sys = new SystemPolicy($OBJECTNAME.text, $PLACEMENTNAME.text);}
    ;
```

4.2.6 Access Policies

Access policies are created both for locations (4.4 on page 44) and data objects (4.8 on page 45). The methods for creating these access policies can be viewed below in listing 4.11 on page 47. The flow works as follows;

accessPolicies creates a `HashSet` of one or more *accessPolicy*s, and returns it.

accessPolicy Divides the line contents into a name and a list of *policyActions*, and creates a new `AccessPolicy`.

policyActions Identifies each *action* in a comma-seperated list

action Returns the appropriate `Action`.

Listing 4.11: Methods for creating and return a list of access policies

```
accessPolicies returns [HashSet<AccessPolicy> policies]:
    {policies = new HashSet<AccessPolicy>();}
    policy1=accessPolicy {policies.add($policy1.ap); }
    (
        policy_other = accessPolicy
        { policies.add($policy_other.ap); }
    )*
  ;

accessPolicy returns [AccessPolicy ap]:
    ACCESSER = NAME ':' actions = policyActions ';'
  {
    ap = new AccessPolicy($ACCESSER.text, actions);
  }
  ;

policyActions returns [HashSet<Action> actionSet]:
    {actionSet = new HashSet<Action>();}
    action1= action { actionSet.add($action1.act); }
    (
        ','
        action_other = action
        { actionSet.add($action_other.act); }
    )*
    ;

action returns [Action act] :
  'i'
  { act = Action.in; }
  | 'o'
  { act = Action.out; }
  | 'm'
  { act = Action.move; }
  | 'e'
  { act = Action.eval; }
  | 'r'
  { act = Action.read; }
  ;
```

4.2.7 Basic Rule Set

We have defined the basic rule set in listing 4.12 on page 48. Here, we define the following.

A *NAME* can contain any letter, as well as the underscore, the asterix or even numbers.

An *INT* can be one or more of numbers between zero and nine.

Spaces, tabs, returns and newlines are all marked for the *HIDDEN* channel. This way, they shall not be interpreted by the parser.

Listing 4.12: Basic Rule Set

```
NAME  :   ('a'..'z'|'A'..'Z'| '_' |'*') ('a'..'z'|'A'..'Z'|'0'..'9'|'_↩
    '|'*')*;

INT :   '0'..'9'+ ;

WS  :   (' ' | '\t' | '\r' | '\n')+ {$channel=HIDDEN;};
```

Chapter 5

Results

In this chapter, we will cover the results gathered from the analysis and policy invalidation done by the tool on three different scenarios. These system models are explained in the design chapter, section 3.6 on page 29. For each of these scenarios, we will document and evaluate the results; is it what we expected?

5.1 Scenario 1

The scenario 1 results for the user, *U*, can be seen in table 5.1 on page 50. We see that no restrictions for reaching the outside location exist. This makes sense, as it is the user's starting location. Entering and passing the *entry* node requires facial recognition, meaning the user's own identity, *U*, is set as a restriction. Any actor can reach the hall, adding no new restrictions. This holds for the exit location as well, as *U* is already in the restrictions. Moving to either the *svr* or *usr* will cross a lock requiring the *code_ U*, which is added to the restrictions for any branches of this. As the *secret_file* is situated on the *pc2*, reaching and picking this up will have the restriction [*U, code_ U*]. In addition to these results, the user has also broken a policy, as shown in listing 5.1 on page 50.

These results yield no surprise for us, as we expect the *U* to invalidate this policy. Furthermore, the restrictions for each location harmonize with the access policies in the scenario - the actor correctly gains the proper restrictions, when passing through locks.

Listing 5.1: U invalidated policy

```
grammar PolicyInval;
Policy with;
Object name: secret\_file
Placement name: outside
has been proven invalid by actor U
```

Locations	Restrictions
outside	null
entry	U
hall	U
lock_usr	U, code_U
lock_svr	U, code_U
exit	U
svr	U, code_U
pc2	U, code_U
usr	U, code_U
pc1	U, code_U
Known Data	**Restrictions**
secret_file	U, code_U

Table 5.1: Results for the actor *U* scenario 1

The *janitor*'s results for scenario 1 can be viewed in 5.2 on page 51. As with the user, we can see that the janitor *J* correctly inherits restrictions when moving between locations with access policies. Furthermore, he doesn't intrude anywhere, which he isn't supposed to reach, based on his knowledge. As opposed to the user, the janitor doesn't break any system policy, as he, despite being able to reach *svr*, doesn't have the permission to access *pc2* and read the *secret_file*.

5.2 Scenario 2

The scenario 2 results for the user *U1* can be viewed in table 5.3 on page 51. As with *U*'s results from scenario 1, in section 5.1 on page 49, *U1* starts up with no restrictions in the starting location *outside*, but sequentially picks up restrictions when passing through *entry* and the *lock_ usr*. The user *U1* is also

Locations	Restrictions
outside	null
entry	J
hall	J
lock_jan	J, key_J
lock_svr	J, code_J
exit	U
svr	J, code_J
jan	J, key_J

Table 5.2: Results for the actor *J* scenario 1

able to pass through *pc1* to the *server* in the virtual domain, invalidating the system policy in 5.2 on page 51.

Locations	Restrictions
outside	null
entry	U1
hall	U1
lock_usr	U1, code_U1
exit	U1
usr	U1, code_U1
pc1	U1, code_U1
server	U1, code_U1

Table 5.3: Results for the actor *U1* scenario 2

Listing 5.2: U1 invalidated policy

```
Policy with;
Object name: U1
Placement name: server
has been proven invalid by actor U1
```

In table 5.4 on page 52, the results for the user *U2* in scenario 2 are shown. These are very similar to those of *U1*, except that *U2* interacts with *pc2* instead of *pc1* and isn't allowed to reach the *server*, which would break a system policy.

Table 5.5 on page 52 shows the results for the user *U3* in scenario 2. As opposed to *U1*, *U3* is permitted to physically gain access to the *server* through the *svr*. As with the previous users, the restrictions are as expected.

Locations	Restrictions
outside	null
entry	U2
hall	U2
lock_usr	U2, code_U2
exit	U2
usr	U2, code_U2
pc2	U2, code_U2

Table 5.4: Results for the actor *U2* scenario 2

Locations	Restrictions
outside	null
entry	U3
hall	U3
lock_svr	U3, code_U3
exit	U3
svr	U3, code_U3
server	U3, code_U3

Table 5.5: Results for the actor *U3* in scenario 2

5.3 Scenario 3

This section contains the results from scenario 3. Here, only one user, *U*, is active, but the scenario is a bit more complicated, as *U* needs to pick up several data objects to invalidate the system policy. From the results in table 5.6 on page 53, we see that *U* correctly has the restriction *U, code_A*, when reaching *usr* and picking up *code_B*, and also giving this data element the proper restriction. When picking up the *code_C* in the *off*, we see that the tool has successfully carried the restrictions of, not only the location, but also of the data `code_B`. Thus, we have the restriction of picking up *code_C* as *U, code_B, code_A*. From here on, the restrictions are all passed down correctly, as *U* moves into *sto* and invalidates the policy shown in listing 5.3 on page 52. Again, the tool performed well, even when facing a more complicated scenario.

Listing 5.3: U invalidated policy

```
Policy with;
Object name: U
Placement name: sto
```

```
has been proven invalid by actor U
```

Locations	**Restrictions**
outside	null
entry	U
hall	U
lock_usr	U, code_A
exit	U
usr	U, code_A
lock_off	U, code_B, code_A
lock_svr	U, code_B, code_A
svr	U, code_B, code_A
off	U, code_B, code_A
lock_sto	U, code_B, code_A, code_C
sto	U, code_B, code_A, code_C
Known Data	**Restrictions**
code_B	U, code_A
code_C	U, code_B, code_A

Table 5.6: Results for the actor U in scenario 3

Chapter 6

Discussion

In this chapter, we evaluate each scenario and their results, as well as the tool for policy invalidation itself. Then, we will give real-life examples of how the tool can be used, followed by a small discussion on formalising problems. At the end of the chapter, we will cover the possible extensions to the tool.

6.1 Evaluation

In this section, we shall evaluate the scenario results in chapter 5 on page 49, as well as evaluate the tool as a whole, covering its behavior and limitations.

6.1.1 Scenarios

The results from scenario 1, section 5.1 on page 49, confirm that the tool can correctly analyse and invalidate policies in a basic scenario. We see that the U actor can pass through appropriate access policies and eventually pick up the *secret_file*, allowing him to break the system policy. The janitor, J, on the other hand, does not break the system policy, despite being able to enter the *svr*.

In scenario 2, section 5.2 on page 50, we see how the tool handles a system model with three different actors, where one is able to break the system policy. The *U3* actor can physically reach the *server*, while neither *U1* or *U2* can. Nevertheless, *U1* can move through the virtual domain from *PC1* to the *server*, thus breaking the system policy. *U2* cannot break this policy, as he lacks permission to enter the *server*.

We see a more sophisticated case in scenario 3, section 5.3 on page 52, where the actor *U* starts up with one code and has to discover new codes to reach the *sto* location to invalidate the system policy. Again the tool performs as intended, as *U*, after gaining new knowledge, revisits previously-covered locations to see if any new 'doors' may be opened.

These results confirm the tool performs correctly analyses the scenarios at hand. But now we need to ask ourselves, what do these scenarios actually cover? Scenario 1 yields a simple scenario, where we want to prove the basic functionality of the tool. Scenario 2 was a different kind of scenario, as we sought to investigate the movement in the virtual domain. The third and final scenario, scenario 3, was a more complex scenario, where multiple data pick-ups was involved, which also tested the tool's ability to reset the actors behavior when getting new knowledge. But something in common for all these scenarios is their size, as none of them involve bigger systems. The reason is that we want to keep things less complex and yield a proof-of-concept of the tool's validity, but it still limits how much the tool can display its potency. Furthermore, any misbehavior only present in larger system models may thus not be caught in this stage. To summarize; even though the chosen scenarios yield good cases for proof-of-concept, any future work on this tool would benefit greatly on applying the tool on larger and more complex scenarios.

6.1.2 The Tool

The results, chapter 5 on page 49, showed us that the tool performed correctly and returned the expected results for each of the three scenarios. The restrictions of locations and data items identified what kind of clearance each actor used to reach or pick-up these, while any system policy was properly declared invalid.

In terms of speed, the tool does not use much time to process any of the scenarios - about the time it takes for you to blink. As mentioned before, the scenarios are not large, but it is estimated that larger scenarios with more branches and opportunities for learning new data will require significantly longer time to process. One of the reasons is that an actor will have to revisit his reachable locations every time, he learns new data. If the scenario contains hundreds of locations with dozens of data items, the amount of re-iterations suddenly increase dramatically.

Despite the usefulness of the implemented tool, it possesses numerous limitations. These limitations occur, because the main goal of the project was to implement a tool to perform system model analysis and policy invalidation, and sub-concepts were less prioritised. We will now cover each of these limitations and explain, what their implications mean for the tool.

The domain name of the class `Location` was never utilised in the `Analyser`'s algorithm. The idea was to support system policies of the form `ACTORNAME !@ DOMAINNAME`, which would translate into restricting the actor from entering any location of that domain.

Of the actions defined in the `Action` class, only the *MOVE* and *READ* actions are acted upon in the analysis algorithm. Even though the most important actions are thus covered, it would make the scenarios, and thus the tool, more life-like, if these commands and appertaining consequences were included in the tool.

The current `Analyser` cannot guarantee a reachable location to be annotated with the least restriction in a system model with cyclic paths for an actor. The reason is that an actor's reachable location is annotated with a restriction the first time, he reaches it. Since the `Analyser` by nature only revisits locations, when an actor learns new knowledge, it is not able to guarantee a location its least possible restriction, if multiple adjacent paths can reach it. Furthermore, an actor's reachable location may only be annotated with one restriction, which would also prove diminishing for the results, if the system model is cyclic. A solution to this could be allowing an actor's reachable location to have more than one restriction.

6.2 The Tool in Practice

In this section, we will devote ourselves to present real-life examples, where this tool will be able to help. These examples are listed below.

A company has just expanded, hiring dozens of new employees. Suddenly, the infrastructure of the company has exploded with new access policies and employees with different sets of rights. Now, they want to make sure that the infrastructure is as safe as possible after the new expansion, as they seek to follow the principle of least privilege. They use the policy invalidation tool, creating numerous system policies tailored to ensure the safety of the company secrets, and reducing the amount of weak links in the infrastructure.

A company has just got crucial information leaked, and they are determined to discover the source of this. The problem is that the company contains thousands of employees in different departments with this crucial information, so person interviews and investigations will take a lot of time and resources. The company learns about the useful tool, which can provide analysis and policy invalidation of system models. Using this tool, they define the appropriate system policies to monitor which actors can interact with the leaked information, and how. Eventually, the tool has helped the company to narrow down the search for the suspect employees.

A couple of friends have just graduated from DTU, and they have planned to start up their own company. They have a great idea for a product, and they have already got the necessary sponsor money to create a department of a dozen of programmers and other employees. They have already created a template for the company's infrastructure, including all newly-hired employees, as well as the basic access control mechanisms, but now they want to evaluate this structure before constructing it physically. The policy invalidation tool helps them test their template infrastructure, as it shows them how it handles different situations of renegade insiders or oblivious managers. This information is used to improve their template infrastructure, and they ultimately create the foundation of their company.

6.3 Formalising Problems

In this project, we have worked on formalising the problem of insider threats and in-secure infrastructures inspired by the *acKlaim* calculus. In the right hands, and in the appropriate cases, formalisation of methods can be used to analyse and verify systems, be it software, hardware or physical systems, as in this project. Despite this being true, it doesn't come without a price [But]. The complexity of today's systems makes the appertaining formalisation equally complex, which is why formalisation of problems are mostly used in safety-critical or high-integrity systems, such as airplanes, where an error may results in human casualties or huge financial losses.

As the systems of today become larger and more complex, it will be exciting to see how this field of technology will adapt for the sake of analysing and verifying the safety in such systems.

6.4 Future Work

In this section, we talk about the future work on this project and the field of technology as a whole. We start of presenting a list of possible extensions to boost the usefulness and precision of the policy invalidation tool:

Extending the tool with **logging** information would not only make it more precise, but also more realistic. In many modern systems, you have logging on some joins in a system model, for example when passing certain locks or on computers. This can prove useful in after-the-math investigations, especially if combined with the element of time.

Adding the element of **time** to the tool would be a massive extension. This would involve adding time estimation attributes to actions, and even allow for system policies to indicate a certain time spam, where the policy would be invalidated. Combining the element of time with logging would improve the quality of the tool even further.

A more trivial, but beneficial, extension would be utilising the **domain** attribute of a location for policy invalidation. This is mentioned in the evaluation of the tool, section 6.1.2 on page 56.

Another improvement in the category of system policies would be to allow `ANYACTOR !@ LOCATION`, meaning that no actor must be allowed to enter that location.

A limitation to the tool is the narrow use of **available actions**, also mentioned in section 6.1.2 on page 56. By adding support for *EVAL*, *IN*, and *OUT*, the larger range of available actions would increase the precision and results of the tool.

A more complex extension is the addition of **encryption/decryption**. With this extension, one would be able to investigate the effectiveness of different encryption protocols, as well as simulate how insiders would go about to break these.

As the policy invalidation tool is inspired by the process calculus, *acKlaim*, it would be beneficial to harness the syntax in order to model workflows. In the current implementation, actors already behave as a process; they move around, perform actions and modify the environment. Due to the similarities, we can treat an actor like we would treat, for example, a package moving from the entrance of an office building to the appropriate receiver. With the addition of the extensions above, this would make the tool very powerful, as it would not only simulate and analyse the movement of actors, but also the flow of items.

These are just some of many possible extensions to the tool, which would increase functionality and precision. If this tool was to be used in real life scenarios, it would, not only have to implement several of the above-mentioned extensions, but constantly keep with with the development of the field of security. That is the eternal battle; the struggle between those who secure, and those who break. As we find new ways to secure ourselves and close the leaks, hackers may find another way to circumvent and disrupt our security measures. It is thus the goal of this type of tool to keep up with the changes in the realm of security, be it in terms of access control, cryptography or cryptoanalysis.

Chapter 7

Conclusion

We have presented the semantics of an analysable system model, formalised using acKlaim, capable of modelling real-world systems. In order to represent these real-world systems, we have designed three different system model scenarios, each with the purpose of showing how various cases are handled. To to read in these scenarios, an interpreter was created in ANTLRWorks, following the system model semantics. On top of this interpreter, we have built a tool for performing analysis and policy invalidation any input scenario read with the interpreter.

The results from the analysis and policy invalidation confirmed what we expected of the tool; it correctly handled several different cases by appending the right restrictions to an actor's reachable locations, as well as invalidating any system policy when appropriate. Despite this success, we argued that the chosen input scenarios may be enough for proof of concept, but larger and far more complicated scenarios should be analysed, if this tool is to be used for academic or professional analysis. In respect to this, several extensions were listed and argued to resolve the current limitations of the tool, making the tool able to analyse far more realistic system models, where such things as encryption/decryption, the time aspect, more available actions, and logging would be beneficial. Even more, due to the nature of our semantics, inherited from acKlaim, the tool can be extended to model workflows, which also would yield a tool capable of modelling far more components in daily life.

As long as systems exist, we will have to deal with the threat of insiders. But as systems grow more complex and insiders become smarter and more innovative, we have to focus on keeping up to par with the latest movement in access control and other types of security measures. Preventive measures, such as building more robust systems and analysing them before-hand using static analysis, have become increasingly popular in order to reduce the potential damage sustained in a system.

We believe that the type of tool developed in this project, formalising real-world systems and using static analysis to extract security-evaluations, will continue to develop, as the need for these kind of measures will increase in parallel with the world itself becoming more complex, and people need to ensure the integrity and validity of their safety-critical products and structures now, more than ever.

Appendix A
Appendix

Here in the appendix, we will cover subjects not crucial for the report. This includes an undisrupted version of the grammar language, as well as the pure textual representation of scenario 1.

A.1 Using the Tool

In order to use the tool, execute the following command in the *bin* directory:

Unix java -cp .:antlr-3.2.jar Main ../input/scenario1.txt

Windows java -cp .;antlr-3.2.jar Main ../input/scenario1.txt

This command runs the `Main` class, appended with the appropriate ANTLR jar, and scenario 1 as the input system model. One can choose between *scenario1.text*, *scenario2.txt*, and *scenario3.text*, or even alter any of these to a slightly different scenario. When altering the scenarios, one must ensure that the syntax can still be confirmed by the modelling language, section 3.2 on page 21.

A.2 Modelling Language Grammar

In this section, we introduce the whole, undisrupted grammar file, see listing A.1 on page 64.

Listing A.1: Grammar file

```
grammar PolicyInval;

@header {
package output;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import dom.*;
}

@lexer::header
{
    package output;
}

@members
{
    private Analyser analyser;
}
//Method for creating and returning the Analyser
getAnalyser returns [Analyser an] :
    {
        HashMap<String, Location> locations = new HashMap<String,
Location>();

        List<Actor> actors = new ArrayList<Actor>();

        List<SystemPolicy> systemPolicies = new
ArrayList<SystemPolicy>();
    }
    'locations'
    '{'
        (
            l = location
            {
                locations.put(l.getName(), l);
            }
            ';'
        )*
    '}'
    'edges'
    '{'
        (
            CURLOC = NAME
            '->'
            exits = exitLocations
            //Add exits to the CURLOC
            {
                Location tempLoc;
                for(int i = 0; i < exits.size();i++)
                {
                    tempLoc = locations.get(exits.get(i));
                    locations.get($CURLOC.text).addExit(tempLoc);
                }
            }
            ';'
        )*
    '}'
    'actors'
    '{'
        {Actor a;}
        (
            ACTORNAME = NAME
            '@'
            STARTLOC = NAME

            {
                a = new Actor($ACTORNAME.text, $STARTLOC.text);
                actors.add(a);
            }
            ';'
        )*
    '}'
    'data'
    '{'
        {Data tempData;}
        (
            DATANAME = NAME
            (
                '{'
                    dataPolicies = accessPolicies
                '}'
            )?
            {tempData = new Data($DATANAME.text, dataPolicies);}
            '@'
            DATALOCATION = NAME
            {
                //If the Data is placed at a Location
                if(locations.containsKey($DATALOCATION.text))
                {

locations.get($DATALOCATION.text).addData(tempData);
                }
                else //Else, the data is placed at an Actor
                {
                    for(int i = 0; i < actors.size();i++)
                    {

if(actors.get(i).getName().equals($DATALOCATION.text))
                        {

actors.get(i).addOwnedData(tempData);
                        }
                    }
                }
            }
            ';'
        )*
    '}'
    'systemPolicies'
    '{'
        (
            sysPol = systemPolicy
            {
                systemPolicies.add(sysPol);
            }
```

```
                ';'
            )*
        '}'
        {
            //Creates a new specification analyser with input locations, ↩
                actors
and system policies
            an = new Analyser(locations, actors, systemPolicies);
        }
        ;

//------------------------------------------------------------------
//Supportive methods goes here

location returns [Location l] : // a location
        LOCNAME = NAME
        ('{'
            policies = accessPolicies
        '}')
        '('
            DOMAINNAME = NAME
        ')'
        {
          // A location is created with ID, set of attributes and set of ↩
              policies
          l = new Location ($LOCNAME.text, policies, $DOMAINNAME.text);
        }
    ;

accessPolicies returns [HashSet<AccessPolicy> policies]:
        {policies = new HashSet<AccessPolicy>();}
        policy1=accessPolicy {policies.add($policy1.ap); }
        (
            policy_other = accessPolicy
            { policies.add($policy_other.ap); }
        )*
    ;

accessPolicy returns [AccessPolicy ap]:
        ACCESSER = NAME ':' actions = policyActions ';'
    {
        ap = new AccessPolicy($ACCESSER.text, actions);
    }
    ;

policyActions returns [HashSet<Action> actionSet]:
        {actionSet = new HashSet<Action>();}
        action1= action { actionSet.add($action1.act); }
        (
            ','
            action_other = action
            { actionSet.add($action_other.act); }
        )*
        ;

exitLocations returns [ArrayList<String> ex] :
        {ex = new ArrayList<String>();}
        firstLoc = NAME
        {ex.add($firstLoc.text);}
        (
            ','
            otherLoc = NAME
            {ex.add($otherLoc.text);}
        )*
        ;
```

```
action returns [Action act] :
  'i'
  { act = Action.in; }
  | 'o'
  { act = Action.out; }
  | 'm'
  { act = Action.move; }
  | 'e'
  { act = Action.eval; }
  | 'r'
  { act = Action.read; }
  ;

systemPolicy returns [SystemPolicy sys] :
    OBJECTNAME = NAME
    '!@'
    PLACEMENTNAME = NAME
    {sys = new SystemPolicy($OBJECTNAME.text, $PLACEMENTNAME.text);}
    ;

//---------------------------------------------------------------------------
//Basic ruleset goes here

NAME  :   ('a'..'z'|'A'..'Z'|  '_'  |'*') ('a'..'z'|'A'..'Z'|'0'..'9'|'_↩
    '|'*')*;

INT :   '0'..'9'+ ;

//NEWLINE:'\r'? '\n' ;

WS  :   (' ' | '\t' | '\r' | '\n')+ {$channel=HIDDEN;};
```

A.3 Scenario 1

Here, we present scenario 1 in pure text form, as it is given to the interpreter, in listing A.2 on page 67. The purpose is to give the reader an understanding of the raw input, which we feed to the interpreter.

Listing A.2: Scenario 1 system model as text

```
locations {
    outside {*:m;}(building);
    entry {U:m;J:m;}(building);
    exit {U:m;J:m;}(building);
    hall {*:m;}(building);
    lock_jan {key_jan:m;}(building);
    jan {*:m;}(building);
    lock_usr {code_U:m;}(building);
    usr {*:m;}(building);
    pc1 {U:m;}(virtual);
    lock_svr {code_U:m;code_J:m;}(building);
    svr {*:m;}(building);
    pc2 {U:m;U:r;}(virtual);
}
```

```
edges{
    outside -> entry;
    entry -> hall;
    exit -> outside;
    hall -> lock_jan, lock_usr, lock_svr, exit;
    lock_jan -> jan;
    jan -> hall;
    lock_usr -> usr;
    usr -> hall, pc1;
    pc1 -> pc2;
    pc2 -> pc1;
    lock_svr -> svr;
    svr -> hall, pc2;
}

actors{
    U @ outside;
    J @ outside;
}

data{
    code_U @ U;
    code_J @ J;
    key_jan @ J;
    secret_file {U:r;} @ pc2;
}

systemPolicies{
    secret_file !@ outside;
    secret_file !@ J;
}
```

Bibliography

[ANT] ANTLR. Antlr. `http://www.antlr.org/`.

[Bov] Jean Bovet. Antlrworks.

[But] R. W. Butler. What is formal methods?

[CWP] Rene Rydhof Hansen Christian W. Probst. Analysing access control specifications.

[CWP08] Rene Rydhof Hansen Christian W. Probst. An extensible analysable system model. 2008.

[CWPN] Rene Rydhof Hansen Christian W. Probst and Flemming Nielson. Where can an insider attack?

[FP] Charles P. Fleeger and Shari Lawrence Pfleeger. *Security in Computing*. Prentice Hall, 4 edition.

[RG91] Deborah Russell and G.T. Gangemi. *Computer Security Basics*. O'Reilly, 1991.