Model and analysis of Role-Based Access Control in SELinux using Description Logic

Alan Ashton Dickerson

Kongens Lyngby 2006
IMM-MASTER-2006-20

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-MASTER: ISSN 0909-3192

Abstract

Security-Enhanced Linux (SELinux) is a version of Linux which, amongst other things, supports Role-Based Access Control (RBAC). The use of the access controls in SELinux have proven to be difficult to use and to perform maintenance upon, especially as the system evolves it may be difficult for the system administrators to comprehend the effects of the changes on the access control policy. Development of an analysis tool for RBAC in SELinux is therefore an important goal.

[Chen Zhao and Lin, 2005] discuss how elements of RBAC can be modeled using the Description Logic $\mathcal{ALLQ}$, and demonstrate how a reasoner for $\mathcal{ALCQ}$ can be used for analysis.

The thesis presents a definition of the access controls of SELinux and shows how to formalize these in $\mathcal{ALCQ}$. It introduces rules for use of an automated implementation of a tool that will model most SELinux configurations. It sketches out ways that the reasoner for an SELinux representation in $\mathcal{ALCQ}$ can be used for analysis by invoking queries.

Keywords: Role-Based Access Control, Description Logic, Security-Enhanced Linux, Formal models

Resumé

Security-Enhanced Linux (SELinux) er en afart af Linux, der understøtter Role-Based Access Control (RBAC). Brugen af adgangskontroller i SELinux har vist sig at være svær at benytte og vedligeholde. Specielt som systemet udvikler sig, kan det være svært for system administratorer at forstå de følger, ændringer af adgangskontrolpolitikken kan have. Udvikling af et analyse værktøj af RBAC i SELinux er derfor et vigtigt mål.

[Chen Zhao and Lin, 2005] diskuterer, hvordan elementer af RBAC kan modellers ved hjælp af logikken Description Logic $\mathcal{ALCQ}$ og viser, hvordan et deduktionsprogram kan benyttes til analyse.

Denne afhandling præsenterer en definition af adgangskontrollerne fra SELinux og viser, hvordan disse kan formaliseres i $\mathcal{ALCQ}$. Der introduceres regler til brug af et automatiseret værktøj, der vil modellere de fleste SELinux konfigurationer. Der skitsers forskellige måder at deduktionsprogrammet kan benyttes til analyse af en $\mathcal{ALCQ}$ model af SELinux ved brug af forespørgselser.

Nøgleord: Role-Based Access Control, Description Logic, Security-Enhanced Linux, Formal models(Formelle modeller)

Preface

This thesis was prepared at Informatics Mathematical Modelling at the Technical University of Denmark in partial fulfillment of the requirements for acquiring a Master of Science in Engineering.

The dissertation deals with different aspects of computer security and formal modeling of the security controls found in Security-Enhanced Linux.

The project was completed in the period of 1st of September 2005 through to the 28th of February 2006 under the supervision of Michael R. Hansen and Robin Sharp.

Alan Ashton Dickerson

Kongens Lyngby, February 2006

Acknowledgements

I thank my supervisors, Michael R. Hansen and Robin Sharp for their support and guidance. Your advice and motivation through the period of time this thesis has taken form has been highly appreciated.

I would also like to thank my lovely wife for her undying support especially through the last period of many hours at school and few at home.

Contents

Abstract **i**

Resumé **iii**

Preface **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Motivation 1
1.2 Objective 4
1.3 Case study 5
1.4 Thesis Overview 6

2 SELinux **9**

2.1 General concepts 10
2.1.1 Flask 10
2.1.2 Type enforcement model 11
2.1.3 Role-based Access Control 12
2.1.4 Users 12
2.2 Policy 13
2.3 SELinux syntax 13
2.3.1 TE declarations 13
2.3.2 RBAC declaration 17
2.3.3 Users 19
2.3.4 Security context 20

2.3.5 Macros 21
2.3.6 Conditionals 22
2.3.7 Special keywords 23
2.4 Software team 24
2.4.1 Assumptions 24
2.4.2 Produce code 25
2.4.3 Execute code 28
2.4.4 Read documentation 28
2.4.5 RBAC 29
2.4.6 Users 29
2.5 Summary 30

3 Description Logic 31
3.1 Introduction 31
3.2 Syntax 32
3.3 Semantics 33
3.4 Reasoner - RACER 34
3.5 Software team 35
3.6 Summary 36

4 SELinux to Description Logic 37
4.1 Model formulation 37
4.2 Translation rules 42
4.2.1 TE declarations 42
4.2.2 RBAC declaration 46
4.2.3 Users 47
4.3 Software team 48
4.4 Alternative model formulations 49
4.5 Summary 50

5 Implementation 51
5.1 Design 51
5.2 Lexical analysis 52
5.3 Abstract data model 52
5.4 Parsing 53
5.5 Translation 54
5.6 Testing strategy 58
5.7 Summary 59

6 Verification 61
6.1 Verification Strategy 61

6.2 Verification . . . 62
6.3 Queries . . . 65
6.4 Software team . . . 68
6.4.1 Miscellaneous queries . . . 70
6.5 Summary . . . 71

7 Discussion 73
7.1 Status . . . 73
7.2 Problems . . . 74
7.3 Related works . . . 74
7.3.1 Models . . . 75
7.3.2 Tools . . . 75
7.4 Summary . . . 76

8 Conclusion 79

A SELinux grammar 83
A.1 Flask . . . 83
A.2 TE . . . 84
A.3 RBAC . . . 85
A.4 Users . . . 86
A.5 Constraints . . . 86
A.6 Security Contexts . . . 86
A.7 Conditionals . . . 88

B Source code 89
B.1 Abstract.sml . . . 89
B.2 Lexer.lex . . . 91
B.3 Gram.grm . . . 93
B.4 makeparser.bat . . . 98
B.5 parse.sml . . . 99
B.6 auxiliary.sml . . . 100
B.7 testFunctions.sml . . . 102
B.8 stringFunctions.sml . . . 103
B.9 main.sml . . . 109

C Case study implementation in SELinux 123
C.1 RBAC . . . 123
C.2 Users . . . 123
C.3 nedit . . . 124
C.3.1 nedit.te . . . 124

C.3.1.1 Reduced version 125
C.3.2 nedit.fc 125
C.4 mosml 126
C.4.1 mosml.te 126
C.4.1.1 Reduced version 126
C.4.2 mosml.fc 127
C.5 more 128
C.5.1 more.te 128
C.5.1.1 Reduced version 128
C.5.2 more.fc 129

List of Tables

1.1 Incident and vulnerability report 2

3.1 Syntax for $\mathcal{ALCQ}$ DL 32
3.2 Interpretation of DL 34
3.3 Concrete syntax for RACER 35

4.1 Ignored declarations 38
4.2 Policy snippet 48
4.3 Policy snippet translated to DL 48

5.1 Grammar snippet 54
5.2 The declarations matched on the first traversal of the parsetree . 56
5.3 The declarations matched on the second traversal of the parsetree 57

6.1 Verification of concept axioms involving roles 67
6.2 Query example of determining the domain permissions for mosml_t 68
6.3 Query example of the types authorized for role tester_r 68
6.4 Query example of determining whether a user can execute a program 69
6.5 Question 5 answered for case study user Tom and type nedit_t . 70

List of Figures

1.1 Role hierarchy of a software team 5

4.1 The Semantic Graph representing the policy of Listing 4.1 on page 39 40

6.1 The role assertion graph of the example policy 64
6.2 The concept assertion graph of the example policy 65
6.3 The concept assertion graph of the example policy 66

7.1 Screenshot from Apol 77

Chapter 1
Introduction

In this chapter a motivation for Model and analysis of Role-Based Access Control in SELinux using Description Logicwill be given and a general case study will be introduced. The chapter will be concluded by an overview of the structure of the thesis.

1.1 Motivation

In the world of the Internet, computers are exposed to constant attacks from malicious users. Attacks are performed on both personal computers and servers that provide some service related to the Internet. The Computer Emergency Response Team Coordination Center (CERT/CC) who, among other things, coordinates responses to security compromises, identifies trends in intruder activity and analyzes product vulnerabilities, reports that the number of attacks are rising, as well as the number of reported vulnerabilities, see Table 1.1 on the next page[1].

There are several reasons that these numbers are increasing. *Software complexity* is an obvious culprit when it comes to introducing vulnerabilities in software.

[1] From `http://www.cert.org/stats/cert_stats.html`

Year	Incidents	Vulnerabilities
2000	21,756	1,090
2001	52,658	2,437
2002	82,094	4,129
2003	137,529	3,784
2004	N/A	3,780
2005	N/A	5,990

Table 1.1: Incident and vulnerability report. Note that CERT/CC stopped publishing the number of incidents as of 2004 due to the widespread use of automated attack tools.

Software is both complex to program with a high level of security, but configuring programs to run securely can also pose problems. The fact that computers are gaining more and more connectivity mainly through the Internet is also a reason that incidents are on the rise. Also, *Mobile code* embedded in objects such as Adobe PDF documents, Microsoft Word documents, Java Applets and the like, pose a security risk.

The most common behavior to minimize risks regarding computer security is to limit access to computers, through filtering firewalls for instance, and to keep programs up to date by installing various security patches.

The problem with security patches is the *patch cycle*. Creating a security patch takes several steps before the patch closes the security issue. Specifically, the issue must be discovered, acknowledged, fixed, released, tested and installed. Such a process can take a long time depending on several factors such as the owner of the program, the severity of the issue or the complexity of fixing the issue.

Having a system with all the latest security patches does not imply that the system is secured from successful malicious attacks because the attackers could know of some security vulnerabilities for which no patch has been released. Such a vulnerability is also known as an *Zero-day vulnerability* because attacks can exploit it prior to or on the same day that the vulnerability has been acknowledged. Moreover, even though a patch has been released which deals with a known vulnerability, many successful attacks are being conducted on systems which have not been patched yet because the system administrator did not install the patch.

The above scenario is the motivation for *Security-Enhanced Linux* (SELinux) which isolates running processes on systems into small *program domains* (or sandboxes) with limited permissions. By such separation SELinux seeks to deal with Zero-day vulnerabilities such that if one program is compromised it can not "break out of its sandbox".

A properly configured SELinux system with the latest security patches will be able to run with a higher level of security than that of a system running with only the latest security patches. The disadvantage with SELinux is that the complexity of designing the different program domains and configuring the system is very high. The configuration of SELinux is very detailed so very specific statements can be made regarding the program domains and their permissions. The system administrator can, for instance, configure some files to only be accessed by one type of program and no others, regardless of who the user operating the program is. The typical operations that a system administrator is in charge of, e.g. installing a new program, adding new users, changing file permissions and verification that the operations were successful, can become very time consuming and difficult due to the complexity of the nature of SELinux's configuration. Furthermore, configuring the system, after installing a new program for instance, have no systematic practise since different programs have different permission requirements. This means that the system administrator typically has to go through a trial and error procedure, setting the program's permission and testing, before the program can run without causing access denied errors. Configuration of the system is done by manipulating SELinux's configuration files, collectively called SELinux's security *policy*. Definition of the parts of the policy typically involves using predefined text replacing macros that enables the system administrator to create hundreds of rules with a single line in the policy file. The downside of the different macros is that it makes it difficult for the system administrator to keep a clear view on exactly which permissions have been set for which domains and files, especially when some program domain definitions can contain 40,000 rules or even more.

The massive size of the collective policy file is one motivator for a formalization of a model which can be used to query different aspects of the policy. Such a model should enable the system administrator to gain an overview of the different roles, programs, permissions etc. in the system. But he should also be able to lookup how these different concepts are allowed to interact with each other, such that he will be able to query the behavior of the system to see if it is set up as he expected. The model will only act as a tool for verifying (or rejecting) the believed system behavior as it is still necessary for the system administrator to configure the different programs in a trial and error procedure as a model of the policy can never express notions regarding the behavior of unknown programs.

The background for this thesis has its roots in the article [Chen Zhao and Lin, 2005] which models *Role-Based Access Control* (RBAC) using Description Logic. The article shows how to utilize Description Logic to model a role hierarchy and the permissions and sessions associated with it. The article does not cover the other types of security controls found in SELinux, but defines a structure for modeling RBAC and shows how to make queries regarding the model. This thesis seeks to apply this structure when modeling SELinux.

1.2 Objective

The hypotheses of the thesis is to see if a formal language, Description Logic, can be used to model the security policy found in SELinux and to extract useful data that a system administrator can use to alleviate the difficulties he faces when configuring and maintaining an SELinux system. The main objective is to model the security policy of SELinux, which defines the permissions that the various programs have when they interact with each other and with different files. Furthermore, a set of queries about the model should be created to show the usefulness of the system and to present the structure of representing an abstract question about the policy into queries that is understood by the Description Language.

Some conditions and assumptions are defined:

The security policy of SELinux is usually based on an example policy that comes with the Linux distribution. For that reason, a condition is set that the policy must be based on the example policy.

To have a better overview of the different domains SELinux divides the policy definition into several files. SELinux's `checkpolicy` program collects all the necessary parts of the system and combines it into one complete file referred in this thesis only as the policy (or the policy file). The program performs syntax check and other integrity checks. Besides outputting the combined policy file in clear text, the `checkpolicy` program also outputs the policy as a binary data file for faster use by SELinux, but that file is irrelevant for the purpose set up in this thesis. The policy file must have been created with the `checkpolicy` parser to ensure that the policy is complete and no syntax errors exists.

To restrict the size of this project it has been chosen to focus on ordinary programs and their permissions. This means that *daemon programs* (i.e. background processes) and policy statements regarding these will not be considered. Also, SELinux offers audit logs of denied permissions, but this thesis focuses on the permissions needed rather than which events will be audited.

The approach to solve the objective is strongly rooted in a case study in which the thesis follows and comments throughout the process of creating a policy, creating the corresponding Description Logic model and verifying the contents of that model.

1.3 Case study

This section introduces the case study that will be used throughout the thesis. The case study seeks to be a small but complete example of a system to be implemented in SELinux. The concepts introduced in this section will be further defined in the subsequent chapters.

As an example of a role hierarchy the software team found in [Sandhu et al., 1996] can be used. By defining the roles appropriately the example will be well-suited for use as case study in this thesis.

The software team consists of four roles, *Supervisor*, *Programmer*, *Tester* and *Member*. These roles each have different permission sets, where the supervisor inherits all of the permissions. The programmer and tester roles share the member roles permissions along with their own unique permissions. See Figure 1.1.

- **Supervisor**: Manages the team, can read documentation, edit, view and run code.
- **Programmer**: Produces and compiles code. Reads documentation. Cannot run the code.
- **Tester**: Tests the code and reads documentation.
- **Member**: Can read documentation.

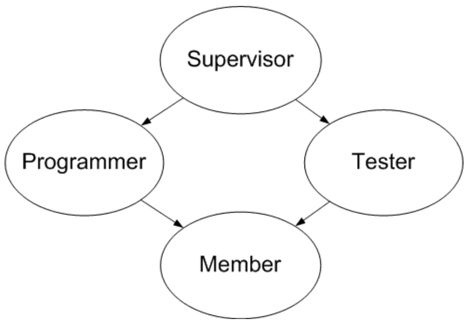


Figure 1.1: Role hierarchy of a software team

The member is the role with smallest permission set. Users assigned to this roles can read documentation and do nothing else. The tester role is in charge of running the code and verifying the programs written by the programmers. The programmer produces code but does not have permission to run it. Both testers and programmers inherit the permissions given to members, meaning that both these roles are able to read documentation. The supervisor role has no permissions by itself, but inherits all the permissions of the subordinate roles.

This means that the supervisor is able to edit, compile and run code and also has authorization to read the documentation.

To instantiate the system, a number of users must be mapped to the roles. The case study will feature one user per role.

- Tom is a Supervisor
- Alice is a Tester
- Bob is a Programmer
- John is a Member

The above syntax implies that the user Tom is a supervisor and so on. The system defines the structure of a software team. It may seem far fetched, that the programmer can not test the produced programs, or that the documentation must already be in place since no one can create it. It is, however, good to show a simple example of the setup and one of the benefits will be to show that such a system can be implemented. Furthermore, it is not hard to expand upon the system to grant more permissions.

1.4 Thesis Overview

The structure of the thesis is as follows:

Chapter 2, *SELinux*, introduces SELinux and the concepts that it uses. The concepts defined there will be used throughout the thesis. The syntax and meaning of SELinux's security policy is detailed and the case study is implemented using those declarations.

Chapter 3, *Description Logic*, brings an overview of the notions of Description Logic forward by going through the syntax and semantics. It also goes through examples to illustrate the syntax of the language and goes through notions of reasoning. The case study is investigated with regard to Description Logic.

Chapter 4, *SELinux to Description Logic*,, takes the two preceding chapters and tries to model the declarations of SELinux in Description Logic. That is done by defining translation rules for every declaration that is necessary in accordance with the thesis objective. It is shown how to use the translation rules using a small subset of the case study. Alternative model formulations are also discussed.

Chapter 5, *Implementation*, focuses on the implementation of an automated tool that inputs any SELinux policy that conforms with the conditions set by the objective and uses the translation rules defined in Chapter 4 to produce a Description Logic model.

Chapter 6, *Verification*, deals with checking if the translated declarations models the actual declarations. Furthermore, a catalog of queries are introduced to answer some questions that a system administrator might have about the policy. Finally, some of the queries are used on the case study's model.

Chapter 7, *Discussion*, discusses the issues that have been found in the translation of the security policy and also refers to other related works.

Chapter 8, *Conclusion*, summarizes the project in a conclusion and makes status upon the objective and looks ahead to suggest items of relevance that could be looked into regarding modeling the SELinux security policy in Description Logic.

Chapter 2

SELinux

This chapter introduces Security-Enhanced Linux (SELinux) and the concepts used regarding it. The concepts and sets introduced are used throughout the thesis. The security policy that defines how SELinux works is detailed and the syntax is informally explained. Furthermore, a security policy addition that implements the case study is defined.

Since SELinux builds on a number of computer security technologies and concepts, a brief overview is given about these but only in relation to SELinux. A good deal of literature about SELinux exists. This thesis have primarily used the book [McCarty, 2004] and the technical report [Smalley, 2005] regarding the security policy syntax and configuration of the policy. It should be noted though, that it was necessary to look up the source code for SELinux's own security policy parser[NSA, 2005] since the grammar detailed in [Smalley, 2005] did not entirely conform with that of the actual parser.

2.1 General concepts

SELinux is an attempt by NSA[1] to create an addition to the traditional Linux kernel that will enable a higher level of security. SELinux has been designed to prevent a wide range of threats on a system:

- Unauthorized access and modification of files and folders
- Unauthorized access to programs
- Circumventing security mechanisms
- Tampering with other processes
- Privilege escalation
- Information security breaches

SELinux accomplishes these goals by extending the existing Linux security model and by introducing a *security server* embedded into the Linux kernel. The security server monitors all actions in the system and checks that they are allowed against a security policy. The default security mode in Linux is based on *Discretionary Access Control* (DAC). This kind of security mechanism relies on the active user's identity. It specifies that programs run by a certain user runs with that users' access rights. Such access control scheme becomes an issue if breaches are made on programs running within a privileged users' access space (typically root). Security-Enhanced Linux implements *Mandatory Access Control* (MAC) using an architecture called *Flask* in the Linux kernel [Loscocco and Smalley, 2000]. Flask enables the administrator to configure the system in a much more detailed manner. SELinux builds upon a modified form of *Type Enforcement* (TE) [Boebert and Kain, 1985] and *Role Based Access Control* (RBAC) [Sandhu et al., 1996].

2.1.1 Flask

Flask security architecture in SELinux requires that every subject (process) and object in the system (file, folder, socket, etc.) is associated with a *security context*. The security context is a collection of security attributes that enables the security server to make decisions regarding collaboration typically between a subject and an object. Internally, the security server uses a *Security Identifier*

[1]National Security Agency in USA

(SID) to identify the security context for flexibility and performance. The SID is an local and nonpersistent integer that is mapped to a security context at runtime. Since it is possible to map an SID to a security context, it is more beneficial to discuss security contexts rather than SIDs. There are two kinds of decisions that the security server has to make: *Labeling decisions* and *access decisions.*

The labeling decision is also known as a transition decision and occurs whenever a process creates a new file or a process starts another process. The security server will need to know what kind of security context the new file or process will have. The transition must be allowed and this is determined by an access decision.

An access decision is based on the concept that every program runs within a *domain*, or *sandbox*, which has a set of *permissions* regarding which files and programs the domain has access to. An access decision will either allow or deny an action based on the permissions (or operations) that are related to the pair of interacting security contexts. Such decisions occur whenever a subject and an object interacts, such as a process wants to read a file or a process wishes to spawn a new process.

The permissions are divided into *security classes* where the classes can be viewed as a name wherein a set of permissions lie, e.g. a create permission associated with a `file` class can be differentiated from a create permissions associated with a `directory` class.

2.1.2 Type enforcement model

SELinux's TE model binds a security attribute called a *type* to every subject (process) and object (files and the like). The type can therefore either represent a domain in which subjects reside or objects. The TE model treats all subjects in the same domain identically as it does with objects of the same type. Access and transition decisions in SELinux are based on a type pair and the security class from the Flask architecture. This enables SELinux to differentiate between objects with the same type but different security classes.

Transition decisions are handled by the TE model to fit the Flask architecture. This means that a transition decision for a process is based on the current domain of the process and the type of the program it wants to execute. The transition rule for an object specifies the new type based on the domain type, the type of the related object and its security class. If no explicit transition rules exist, SELinux uses default rules in accordance to the security class. For a process this means that it retains the domain upon execution. For objects the type of the related object is used for the new object, e.g. files receives the type of the parent folder.

An access rule specifies an allowance based on the type pair and the security class. If no rules matches the pair of types in the security class access is denied. For use of tuning the security policy and performing forensic investigation, an access log is kept consisting of denied access attempts. It is also possible to use access rules to specify auditing rules, i.e. to audit when access is granted or not to audit access denials.

Note that this section did not discus the differences between the traditional TE model and the one used in SELinux, consult [Smalley, 2005] for those details.

2.1.3 Role-based Access Control

Traditional RBAC is a useful division between users and permissions. RBAC introduces roles, permissions limited to that role and authorization of users to the roles. A *role* in SELinux is a group that have been authorized to a set of TE domains. SELinux binds roles and types together instead of roles and permissions as the traditional role based access controls does, as a result of the TE model.

The role based access controls of SELinux consist of methods for creating roles and authorizing sets of types to these roles. It is also possible to set up a hierarchy of roles and wherein type authorization is inherited.

2.1.4 Users

SELinux's user model uses a separate *user identity* to make the MAC of SELinux independent of the existing Linux access control (DAC). This means that the users have to be declared separately. This is done by declaring the user and the roles which the user is authorized for.

2.2 Policy

The SELinux policy consists of a number of declarations that defines the overall security policy of the system. SELinux comes with an example policy that in most cases is used as a starting point for system administrators who customize this policy to fit their needs. Policy creation is, or rather should, be based on principle of least privilege. Since SELinux denies access unless an explicit access rule exists it is a good platform to implement the principle of least privilege.

2.3 SELinux syntax

The policy is expressed in a language with a context-free grammar and the complete set of production rules can be found in Appendix A on page 83. A simple syntax is presented here for the noteworthy declarations to illustrate the typical syntax along with examples of their uses.

The following sets are identified for use throughout the remainder of the thesis: A_{SELinux} is the set of attributes of the system, T_{SELinux} represents all the types in the system, and D_{SELinux} is the subset of types that are domains. R_{SELinux} is the set of all roles in the system and U_{SELinux} represents all the users defined by SELinux's policy. P_{SELinux} is the permissions in the system and C_{SELinux} is the security classes. In SELinux a permission is related to a pair (permission,class), e.g (`entrypoint`,`file`) is the permission that a new domain can be entered with a file (program) or (`write`,`directory`) which is the permission to write upon directories. It is worth to note that not all pairs of permissions and classes makes sense, e.g. (`entrypoint`,`directory`) makes no sense since a directory cannot enter a domain. Therefore the set of all (sensible) permissions is PC_{SELinux} where $PC_{\text{SELinux}} \subset P_{\text{SELinux}} \times C_{\text{SELinux}}$.

The full syntax is not explored here as well as some declarations are omitted. Relating the examples is not a priority as much as it is to emphasize the syntax, so a connected example is not shown, look for this in Section 2.4 on page 24. The simple syntax features keywords in `verbatim` and identifiers appears in *italic*.

2.3.1 TE declarations

The type enforcement declarations are the main components of the policy and deal with declaring the types of the system and the way the types interact.

Attributes

An attribute is a name that identify a set of types with a similar property. Attributes can be used when declaring a type and can be associated to any number of types. The attributes can be used when specifying type allowances, see Section 2.3.1 on the next page, but not to represent part of a security context, see Section 2.3.4 on page 20.

`attribute` *Attributename*;

Examples:

```
attribute domain;
attribute file_type;
attribute privuser;
```

The above example represents declarations of three attributes that can be used to denote a set of domain types, filetype types and privileged user types.

Type declaration

A type in SELinux represents the last part of the security context and can be associated with the domain attribute which means that a process can run within that domain. Types can be declared with any number of attributes and afterwards used to specify the permission set and action when one type interacts with another.

When declaring a type, a type name is needed and if the type shares a similar property with other types, these can be linked together by assigning a number of attributes that describes this property. It is also possible to specify name aliases for a type using this declaration. Furthermore, two keywords exists that adds aliases and attributes to already existing types: `typealias` and `typeattribute`, consult Appendix A on page 83 for details about these.

Let $\mathit{Attributes} \subseteq A_{\mathrm{SELinux}}$ in

`type` *Typename*, *Attributes*

Examples:

```
type esales_t, domain;
type esales_exec_t, file_type, exec_type;
type new_order_t, file_type;
```

The first type declaration specify a type domain, `esales_t`, in which an e-commerce program will run. The second type will be used for the actual program, since it is a file type and it is an executable. Assuming that the program produces files with each order, the last type declaration specifies a type that will be used for these files.

Type access rules

The type access rules is what lies behind most access decisions that SELinux must make. This kind of declaration tells which kinds of permissions are allowed when two sets of types interacts upon a set of classes. The declaration specifies:

- `allow`: Allows access.
- `auditallow`: The action is authorized but will be audited in the access log whenever it occurs.
- `dontaudit`: Denied access attempts are not audited. Useful if it is acknowledged that a domain attempts to gain access to objects, so those attempts should not appear in the access log.

Forward reference to types is allowed, i.e. types do not have to be defined before they are used. Note that $Types_S \subseteq D_{\text{SELinux}}$ while $Type_T \subseteq T_{\text{SELinux}}$. This is due to the fact the only domains are allowed to interact with other types.

Let $\emptyset \subset Types_S \subseteq D_{\text{SELinux}}$ ($Types_S$ is a non-empty set of types),
$\emptyset \subset Type_T \subseteq T_{\text{SELinux}}$,
$\emptyset \subset Classes \subseteq C_{\text{SELinux}}$ and
$\emptyset \subset Perms \subseteq PC_{\text{SELinux}}$ in

`allow|auditallow|dontaudit` $Type_S\, Type_T : Classes\ Perms;$

Examples:

```
allow esales_t esales_sock_t:tcp_socket
 {ioctl read getattr write setattr append bind connect
 getopt setopt shutdown listen accept};
allow esales_t new_order_t:file {create write};
allow acct_rcv_t new_order_t:file read;
```

The first declaration defines to the security server that when the `esales_t` interacts with the `esales_sock_t` type (presumably an Internet socket the e-commerce program uses) it should allow the operations: `ioctl, read, getattr` etc. when these operations are tried on the `tcp_socket`. The second declaration specifies that the program operating within the `esales_t` domain can create and put data into files of the type `new_order_t`. The final declaration tells the security server that a program running in the `acct_rcv_t` domain (an accounting program) can read the new orders.

Type transition

A type transition sets the result of a labeling decision by specifying a new domain for a spawned process or the new type for an object. For processes, the source type is a process and the target type is the type of the executable. If the transition is regarding objects, the source type is the domain of the creating process and the target type the is type of the object. If there exists no type transition rule between source and target types the security server will label the new files and process according to their parents. For files this means that they will receive the parent folder's type. For new processes it means they will run within the same domain as the creating process.

Let $\emptyset \subset Sourcetypes \subseteq T_{\text{SELinux}}$,
$\emptyset \subset Targettypes \subseteq T_{\text{SELinux}}$,
$\emptyset \subset Classes \subseteq C_{\text{SELinux}}$ and
$Newtype \in T_{\text{SELinux}}$ since $Newtype$ has to be defined somewhere:

`type_transition` *Sourcetypes Targettypes:Classes Newtype*;

Examples:

```
type_transition httpd_t var_log_t:file httpd_log_t;
type_transition httpd_t tmp_t:{file lnk_file sock_file fifo_file
    chr_file blk_file} httpd_tmp_t;
type_transition initrc_t sshd_exec_t:process sshd_t;
```

The first rule states that when the webserver domain `httpd_t` creates a file in the `var_log_t` dir, the file will get the new type `httpd_log_t`. The second rule tells that the classes of files mentioned (`file, lnk_file`, etc.) in the temporary directory created by the webserver domain, will get a specific webserver temporary file type, `httpd_tmp_t`. The final rule shows that when the ssh daemon executable is started at boot time by the `initrc_t` domain, it gets its own domain `sshd_t` type.

Type change

Type change is supported by the policy but not used by the security server. A type change can occur when a security-aware program explicitly requests a type change. The type change declaration then comes into play. This typically happens when system daemons relabel terminal devices in user sessions. The syntax is the same as with type transitions except for the `type_change` keyword.

Examples:

```
type_change user_t tty_device_t:chr_file user_tty_device_t;
type_change sysadm_t sshd_devpts_t:chr_file sysadm_devpts_t;
```

The first declaration is used by the `login` program and specifies a terminal device for the user instead of the normal terminal device. The last declaration creates a pseudo-terminal device for the system administrator from the device that originally was allocated to the `sshd` daemon.

2.3.2 RBAC declaration

To feature RBAC in SELinux, roles are supported by declaring these with a set of types that they are authorized for. Definition of a role hierarchy is also supported.

Role declaration

The role declaration authorizes a role to a set of types. The types have to be domains in order to influence the system. In other words, it makes no sense to authorize a role to a file type for instance.

Let $\emptyset \subset Types \subseteq D_{\text{SELinux}}$ in

$$\texttt{role}\ \mathit{Rolename}\ \texttt{types}\ \mathit{Types};$$

Examples:

```
role system_r types {kernel_t initrc_t getty_t klog_t};
role user_r types {user_t user_firefox_t};
role sysadm_r types sysadm_t ;
```

The above declarations are examples of role declarations and type authorizations. The `system_r` is used for programs run by the system itself and must be authorized to the domains that are run exclusively by the system. The `user_r` role is associated to regular (unprivileged) users and must be authorized for the unprivileged domains. Lastly, the system administrator can access his domain.

Role allowance

A role allowance indicates that a role transition is allowed. Such transitions happens when an external action requests a role transition. This is different from domain transitions which can be specified using the TE policy.

Let $\emptyset \subset Currentroles \subseteq R_{\text{SELinux}}$ and
$\emptyset \subset Newroles \subseteq R_{\text{SELinux}}$ then

$$\texttt{allow}\ Currentroles\ Newroles;$$

Examples:

```
allow system_r {user_r sysadm_r};
allow user_r sysadm_r;
```

The above allowances shows that the `system_r` role is authorized to transition to the user and system administrator role and that the user role can transition to the administrator role.

Role dominance

Role dominance is SELinux's way of defining a role hierarchy. By using role dominance the head role inherits the authorizations to the domains granted the dominated roles. The head role need not be defined previously, as a dominance declaration is enough to declare a role also. Note that the `Dominatedroles` can be empty. In such case, the declaration simply creates a role with no inheritances and no type authorizations.

Let $Dominatedroles \subseteq R_{\text{SELinux}}$ then

$$\texttt{dominance}\ \{\texttt{role}\ \mathit{Headrole}\ \{\texttt{role}\ \mathit{Dominatedroles}\}\}$$

Examples:

```
dominance {role supervisor_r {role programmer_r; }}
```

The example shows that the supervisor inherits the types associated to the programmer role.

Role transition

Role transition is a way of changing the active role based on interaction with certain types. A role transition declaration specifies the new role of a process based on the current role and the type of the executable. Role transition is allowed but is not a preferred way to change permissions as the default behavior of executing an executable is to remain in the same role. The preferred method is to define another domain and transition into it. It is however, useful for automatic role transition when restarting daemons. Otherwise role changes should only occur by explicit request by the user.

Let $\emptyset \subset \mathit{Currentroles} \subseteq R_{\text{SELinux}}$ and
$\emptyset \subset \mathit{Types} \subseteq T_{\text{SELinux}}$ in

$$\texttt{role\_transition}\ \mathit{Currentroles}\ \mathit{Types}\ \mathit{Newrole};$$

Examples:

```
role_transition sysadm_r crond_exec_t system_r;
role_transition sysadm_r ftpd_exec_t system_r;
```

The examples states that if the system administrator runs `crond` or the ftp daemon the role will change to the `system_r` role from the `sysadm_r` role.

2.3.3 Users

The user attribute of a security context is unchanged by default when a transition occurs. This can be overridden by SELinux-aware applications. Such ability is controlled by a constraint configuration in the policy. The users declared in SELinux's policy are independent with the normal user concept of

Linux (i.e. those users found in `/etc/passwds`). In many cases it is desirable to map regular users to a single SELinux user. So if the regular Linux users are not found in the user declarations of SELinux, they will be mapped by default to an unprivileged user, `user_u`.

User declaration

Users are assigned their roles in the system by a user declaration. The user declared is often a real person, but a user entity controlled by the system (`system_u`) is also declared by this declaration.

Let $\emptyset \subset Roles \subseteq R_{\text{SELinux}}$ in

$$\texttt{user}\ \mathit{UserID}\ \texttt{roles}\ \mathit{Roles};$$

Examples:

```
user system_u roles system_r;
user root roles { staff_r sysadm_r };
user johndoe roles user_r;
```

The user identity `system_u` is designed for system processes and object and is assigned the corresponding role `system_r`. The `root` user can either inhabit the staff or the system administrator role. The staff role is less privileged than the system administrator. Finally, `johndoe` is assigned the `user_r` role as is any other Linux users that are not mentioned in the SELinux policy. The difference between the `johndoe` and the `user_u` user is that any files `johndoe` creates will have the security context with his user name appended, any other user will have the anonymous user `user_u` attribute in the security context.

2.3.4 Security context

Every object in SELinux is associated with a security context. Whether it is a file, a directory a socket file etc. In order to add new programs the security administrator initially specifies which objects must have which contexts.

File context

$$\mathit{Regex}\ \mathit{Filetype}\ \mathit{Context}$$

The regular expression matches the filesystem and the filetype is an optional field that tells if security context should only be applied for the specified filetype, e.g. only directories, only files, only socket files etc. The context is the security context consisting of a user identity, a role and a type. The role is only applicable for processes. For files and the like, the `system_u`, "system user", is the user and the generic `object_r` is used as role.

Examples:

```
/usr/bin/nmap -- system_u:object_r:nmap_exec_t
/usr/share/nmap.* system_u:object_r:nmap_t
```

The first rule specifies that the nmap executable should be associated with the default system user and object role but have the type `nmap_exec_t`. The `--` indicates that the filetype is a regular file. The second rule tells the system that all files and directories will be mapped with the default system user and object role, but with the `nmap_t` type.

2.3.5 Macros

Since many of the declarations are going to be very similar, the designers of SELinux made it possible to use a macro language(a *m4* style macro language, [Kernighan and Ritchie, 1977]). A macro can either be used as an alias for a set of classes, permissions etc.(text replacement) or as a string generating function that takes a number of variables.

Macros are used extensively throughout the sample policy to ensure that certain declarations are not left out and to ease the creation of program domains.

The following examples are taken used from the above examples to show how the text replacing macros can enhance the readability of the declarations. First the syntax of the declaration including the macro, thereafter the resulting declaration.

Examples:

```
type_transition httpd_t tmp_t:file_class_set httpd_tmp_t;
->
type_transition httpd_t tmp_t:{file lnk_file sock_file fifo_file
    chr_file blk_file} httpd_tmp_t;

allow esales_t esales_sock_t:tcp_socket { rw_socket_perms listen
    accept};
->
allow esales_t esales_sock_t:tcp_socket
```

```
{{ioctl read getattr write setattr append bind connect getopt setopt shutdown}
    listen accept};

domain_auto_trans(initrc_t, sshd_exec_t, sshd_t)
->
.
.
.
type_transition initrc_t sshd_exec_t:process sshd_t;
```

The above shows examples based on previous declarations. The first two examples uses text replacement, namely with `file_class_set` and `rw_socket_perms`. Since the `rw_socket_perms` does not contain the `listen` and `accept` permission, it is needed to add these.

The final macro operates as a string generation function that takes the arguments and generates declarations according to the function definition. The macro grants the permissions to the `initrc_t` domain to type transition into the `sshd_t` domain when executing the `sshd_exec_t` type. It also allows other permissions, such as granting the `initrc_t` access to read and execute the `ssh` executable file. The macro generates 12 declarations in total and can be seen on Listing 2.1.

2.3.6 Conditionals

The security policy can define a number of booleans that will specify if certain allowances should be granted. The booleans are named expressively and used in conjunction with an *if-then-else* construction that specifies the allowances in case the boolean is false and in the case it is true. Note that it is allowed that the `else` block is left out. The blocks can only consist of type transition declarations or type allowance declarations, see Appendix A.7 on page 88.

`bool` *Booleanname* `true | false`;
`if(` *condition*`)` { *block* } `else` { *block* }

Examples:

```
bool allowaccounting false;
if (allowaccounting)
  {allow acct_rcv_t new_order_t:file read;}
else
  {dontaudit acct_rcv_t new_order_t:file read;}
```

Listing 2.1: Macro expansion of `domain_auto_trans`

```
# Make executing sshd_exec_t enter the sshd_t domain
# Allow the process to transition to the new domain.
allow initrc_t sshd_t:process transition;
# Do not audit when glibc secure mode is enabled upon the
    transition.
dontaudit initrc_t sshd_t:process noatsecure;
# Do not audit when signal-related state is cleared upon the
    transition.
dontaudit initrc_t sshd_t:process siginh;
# Do not audit when resource limits are reset upon the transition.
dontaudit initrc_t sshd_t:process rlimitinh;
# Allow the process to execute the program.
allow initrc_t sshd_exec_t:file { read { getattr execute } };
# Allow the process to reap the new domain.
allow sshd_t initrc_t:process sigchld;
# Allow the new domain to inherit and use file
# descriptions from the creating process and vice versa.
allow sshd_t initrc_t:fd use;
allow initrc_t sshd_t:fd use;
# Allow the new domain to write back to the old domain via a pipe.
allow sshd_t initrc_t:fifo_file { ioctl read getattr lock write
    append };
# Allow the new domain to read and execute the program.
allow sshd_t sshd_exec_t:file { read getattr lock execute ioctl };
# Allow the new domain to be entered via the program.
allow sshd_t sshd_exec_t:file entrypoint;
type_transition initrc_t sshd_exec_t:process sshd_t;
```

The example defines a boolean that tells whether the system administrator wants to allow the use of an accounting program. The construction below will evaluate the condition and use the second block in the security policy meaning that attempts to use the accounting program to read new orders will not cause audit information to be recorded.

2.3.7 Special keywords

All sets used in a policy declaration (except the attribute set of the attribute declaration) can contain special keywords. The first special keyword used is `self`. That keyword indicates that source type is authorized with the succeeding permissions upon itself. If there are more types than one and the `self` keyword appears, the rule is applied between each source type and itself. Due to the nature of this keyword it only makes sense when the type it represents is a target type. There also exists some special characters that can be used in sets, these are the asterisk character (`*`), the tilde character (`~`) and the minus character (-) which are used to denote the set's universe, set complement and set substraction.

```
allow sshd_t self:*  ~{ execute write};
allow sshd_t file_type - var_log_t:file read;
```

The first example states that the `sshd_t` domain when interacting with itself on all security classes, have all permissions except `execute` and `write`. The second example states that the `sshd_t` domain can write to all files (`file_type` is an attribute representing the set of types that are files) except files in the `\var\log` directory.

2.4 Software team

To continue the ongoing example, the time has come to implement the case study in SELinux. First of all, some assumptions must be made since the permissions and domains must be set with regards to actual programs and files. The chosen programs and file structure is based on a simple case study with regard to required permissions. A program like Java, for instance, will need many more permissions than Moscow ML since having a virtual machine demands more permissions.

2.4.1 Assumptions

Code directory:	`/var/code/`
Documentation directory:	`/var/code/doc/`
The editor to produce code:	`nedit`
Moscow ML to execute code:	`mosml`
To view documentation the text viewer:	`more`

The fictive users specified in Chapter 1.3 on page 5 will also be incorporated into the policy.

Creating a domain for a new program is an iterative process, where some steps are taken a couple of times to ensure that the domain and the programs residing in it functions as intended. In an attempt to systematize the process, the following procedure was created.

1. Determine the intended behavior of the domain

2. Create the types to be used by the domain
3. Create matching file context for initial file mapping
4. Create rules to match the intended behavior
5. Create required rules for the program to run
6. Verify behavior by running programs in domain
7. If needed, modify rules to match the required allowances

Typically, most time will be spent on the last three items. Since the intended behavior is set, the challenge is usually determining which permissions are needed for the program to run. Even though the procedure describes the steps necessary to create a program domain, the process is based on trial and error, making it difficult to systematize.

2.4.2 Produce code

To produce code, the editor `nedit` was chosen. The program will run in the `nedit_t` domain and must have permission to create files in the code directory. As stated in Chapter 1.3 on page 5, the programmer is the only role that has access to this domain. The below declarations ensures that such policy will be in effect.

Create the necessary types.

```
type nedit_t, domain;
type nedit_exec_t, exec_type, file_type;
type code_t, file_type;
```

Define the file security contexts. The relationship between the executable type in the domain declaration and the file context declaration is apparent.

```
# nedit program is marked as the nedit_exec_t type
/usr/local/bin/nedit system_u:object_r:nedit_exec_t
# code dir is labeled
/var/code system_u:object_r:code_t
# code files will be labled with code_t (excluding dirs)
/var/code.* -- system_u:object_r:code_t
```

Define the role and set type authorization to the `nedit` domain.

```
role programmer_r types { nedit_t };
```

Use macro to make the role a full user. The macro `full_user_role` generates an immense number of declarations in order to allow the intended behavior, namely that the programmer is a full user, i.e. can login can have a home directory etc. The macro takes care of creating new types to accommodate this property, but it uses many declarations to do so since there are many conditions to take into account, such as the many different programs that saves their user settings into a user's home directory.

```
full_user_role(programmer);
```

Transition to the `nedit` domain when executing the program executable. The macro invoked ensures that the user domain are authorized to access and execute the `nedit` executable. It also takes care of issuing a type transition declaration so the security server knows to enter the `nedit` domain `nedit_t` upon execution of the program.

```
domain_auto_trans(userdomain, nedit_exec_t, nedit_t);
```

Allow that users can log in to the programmer role. Only authorized users will be able to do so, see Section 2.4.6 on page 29.

```
role_tty_type_change(user, programmer);
```

Set the authorizations of the domain to fit the wanted behavior.

```
# nedit can read and create files in the code_t dir
allow nedit_t code_t:file create_file_perms;
allow nedit_t code_t:dir create_dir_perms;

# nedit recursively looks up the code dir, so it must also
# be authorized to to the /var dir
allow nedit_t var_t:dir search;
```

The above declarations shows what has been determined as the intended behavior. Listing 2.2 on the facing page shows what is needed in order to be able to run the program. Those declarations have no real correlation to the intended behavior. It can be seen as a predetermined set of permissions that the system administrator must incorporate into the policy to ensure that the program runs.

It can be noted that the bulk of the policy deals with permissions pertaining getting `nedit` to run, i.e. setting the default permissions that enables it to run. Line 4 and 5 enables the `nedit_t` domain to read and write to the `default_t` dir and files which is where `nedit` has its base configuration files. Line 10 invokes a macro that enables the `nedit_t` domain to access common shared libraries. Lines 30-37 enables `nedit` to get attributes about various programs in the user's home directory, typically to check whether they are installed or not.

Listing 2.2: Required allowances for `nedit`

```
### The below declarations define rules that allow nedit to run

#nedit can read its settings files
allow nedit_t default_t:dir rw_dir_perms;
allow nedit_t default_t:file rw_file_perms;

allow nedit_t programmer_devpts_t:chr_file { getattr read write
    ioctl };

# nedit uses the shared libraries
uses_shlib(nedit_t);

allow nedit_t bin_t:dir { getattr read };
allow nedit_t default_t:dir { getattr read search };
allow nedit_t default_t:file { getattr read };
allow nedit_t etc_runtime_t:file { getattr read };
allow nedit_t lib_t:file { getattr read };
allow nedit_t locale_t:dir search;
allow nedit_t locale_t:file { getattr read };
allow nedit_t self:unix_stream_socket { connect create getattr read
     write };
allow nedit_t proc_t:dir search;
allow nedit_t proc_t:file { getattr read };
allow nedit_t sbin_t:dir { getattr read };
allow nedit_t tmp_t:dir search;
allow nedit_t usr_t:file { getattr read };
allow nedit_t xdm_tmp_t:dir search;
allow nedit_t xdm_tmp_t:sock_file write;
allow nedit_t xdm_xserver_t:unix_stream_socket connectto;
allow nedit_t default_t:lnk_file read;

allow nedit_t user_evolution_home_t:dir getattr;
allow nedit_t user_fonts_t:dir getattr;
allow nedit_t user_gnome_secret_t:dir getattr;
allow nedit_t user_gnome_settings_t:dir getattr;
allow nedit_t user_home_t:dir getattr;
allow nedit_t user_mozilla_home_t:dir getattr;
allow nedit_t user_mplayer_home_t:dir getattr;
allow nedit_t user_thunderbird_home_t:dir getattr;

##### End of nedit
```

For the two remaining domains only the declarations regarding the intended behavior are shown, the rest can be found in the Appendix C on page 123.

2.4.3 Execute code

To execute code, the `mosml` program was chosen. The program will reside in the code directory and must be able to read code files. The program should be available to the testers.

Create the necessary types.

```
type mosml_t, domain;
type mosml_exec_t, exec_type, file_type;
```

Define the file security context.

```
# mosml binary
/var/code/mosml -- system_u:object_r:mosml_exec_t
```

Define the role and use macro to transition into the new domain.

```
role tester_r types { mosml_t };

full_user_role(tester);
domain_auto_trans(userdomain, mosml_exec_t, mosml_t);

role_tty_type_change(user, tester);
```

Set the wanted behavior. The `mosml` will receive read permissions on the code directory and files.

```
allow mosml_t code_t:file r_file_perms;
allow mosml_t code_t:dir r_dir_perms;
```

2.4.4 Read documentation

The last domain to be defined is the one of the `more` program. `more` is used to read documentation and should only be available for user authorized for the member role.

Create the necessary types.

```
type more_t, domain;
type more_exec_t, exec_type, file_type;
type documentation_t, file_type;
```

Define the file security contexts.

```
/bin/more -- system_u:object_r:more_exec_t
/var/code/doc.* system_u:object_r:documentation_t
```

Define the role and use macro to transition into the new domain.

```
role member_r types { documentation_t more_t };

full_user_role(member);

domain_auto_trans(userdomain, more_exec_t, more_t);

role_tty_type_change(user, member);
```

Set the wanted behavior. `more` is allowed to read the `documentation_t` type for files and directories. Furthermore, `more` searches the code directory so it needs to be allowed to do so.

```
allow more_t documentation_t:file r_file_perms;
allow more_t documentation_t:dir r_dir_perms;

# more can lookup in the code dir
allow more_t code_t:dir search;
```

2.4.5 RBAC

The convention when declaring policies in SELinux states that the role declarations should be found near the types which they are authorized for. But if a role hierarchy is to be defined, it is done separate from the domain definitions. The case study's roles and hierarchy are defined by the following `dominance` and `allow` declarations.

```
dominance { role supervisor_r {role programmer_r;  } };
dominance { role supervisor_r {role tester_r; } };

dominance { role tester_r { role member_r; } };
dominance { role programmer_r { role member_r; } };

allow supervisor_r { tester_r programmer_r member_r };
allow programmer_r { member_r };
allow tester_r { member_r };
```

2.4.6 Users

Since the used programs, domains, files and roles have been specified in the policy, what remains to be defined is how users are authorized for the different roles.

```
user Tom roles { supervisor_r };
user Alice roles { tester_r };
user Bob roles { programmer_r };
user John roles { member_r };
```

2.5 Summary

The chapter defined the SELinux and the concepts used. A walk through of the syntax used and the meaning of the different declarations was specified. A procedure was developed in an attempt to systemize the steps needed to create a program domain. It was found that such a procedure can not be explicitly defined since the method of creating the policy is an ad hoc method which differs depending on the program. This observation supports the motivation for this thesis, namely that a model that will help a system administrator to verify the properties of the system.

The declarations of SELinux enables the case study to be verified, this was done in Section 2.4 on page 24. The next step is to introduce the concepts used in Description Logic and to create a system that can model the case study, and in general model SELinux.

Chapter 3

Description Logic

This chapter presents *Description Logic* (DL), its syntax and semantics. Reasoning services are presented and a specific reasoner and its syntax is introduced. The sections couples the theory with examples and the chapter concludes with a brief view of the role hierarchy in the case study modeled in DL.

DL is a relative new formal language derived from the need of knowledge representation. The literature used comes mostly from the Description Logic Handbook, [Baader et al., 2003], but also from the article that modeled RBAC in DL, [Chen Zhao and Lin, 2005].

3.1 Introduction

DL is a formal language that is used for knowledge representation and to model knowledge bases. DL consists of *concepts*, *roles* and *individuals* denoted by unary predicates, binary predicates and objects of the domain respectively. The interpretation of a concept is a set of items that share a similar property denoted by the concept's name. Note that the use of the term roles in DL has a different meaning than that of RBAC as defined in Section 2.1.3 on page 12 where it is used to denote a group of users. Here it is used to denote a binding of concepts.

An individual is an instantiation of one of the items residing in a concept. Concept names and role names are written in Sans whereas individuals appear in *italic* when used specifically, i.e. not in the general terms of Table 3.1 for instance, but in the context of specific assertions, like Equation 3.1 on the next page.

The DL used in this application is the $\mathcal{ALCQ}$ language. It is the basis DL language $\mathcal{AL}$ extended by negation (complement, hence $\mathcal{C}$) and qualified number restriction (denoted by $\mathcal{Q}$).

3.2 Syntax

The language's concept descriptions has the syntax rules showed on Table 3.1.

C, D	$\longrightarrow$	$A \mid$	(atomic concept)
		$\top \mid$	(universal concept)
		$\bot \mid$	(bottom concept)
		$\neg C \mid$	(concept negation)
		$C \sqcap D \mid$	(intersection)
		$\forall R.C \mid$	(value restriction)
		$\exists R.C \mid$	(existential quantification)
		$(\geq nR.C) \mid$	(qualified number restriction)

Table 3.1: Syntax for $\mathcal{ALCQ}$ DL

A knowledge base has two components, a TBox and an ABox. A terminology of concepts and roles are defined which specifies the TBox. It defines concepts and roles through terminological axioms, namely inclusions and equalities. An ABox is the instantiation of the TBox so to speak, meaning that the ABox is assertions regarding individuals in the terminology defined by the TBox.

Concept axioms which builds the TBox have the form:

$$C \sqsubseteq D \qquad C \equiv D$$

$C \sqsubseteq D$ defines an inheritance relationship between the two concepts, also known as an *IS-A* relationship, where C is a *specialization* of D.

Some examples of the axioms:

$$\begin{aligned}\mathsf{AmphibiousVehicle} &\equiv \mathsf{GroundVehicle} \sqcap \mathsf{WaterVehicle}\\ \mathsf{Woman} &\sqsubseteq \mathsf{Person}\\ \mathsf{Mother} &\equiv \mathsf{Woman} \sqcap \exists \mathsf{hasChild.Person}\end{aligned}$$

The examples states that an amphibious vehicle is defined as the conjunction set of ground vehicles and water vehicles. The second axiom states that Woman is a specialization of Person. The last equality axiom states that a mothers are woman that also has the hasChild relation in the Person concept.

c is a filler of the role R for individual b, that has children who are in the Person set.

Individuals are defined by giving them names and asserting properties regarding them using assertions. There are two kinds of assertions that creates the ABox.

$$\mathsf{C}(a) \qquad \mathsf{R}(b, c) \tag{3.1}$$

Concept assertions state that the individual a belongs to concept C. A role assertion states that individual c is a filler of the role R for individual b, see Equation 3.1

Some examples of assertions:

$$\mathsf{Woman}(Mary) \qquad \mathsf{hasChild}(Mary, Paul)$$

The example states that the individual name Mary is a Woman, and that (the individual) Paul is the child of Mary.

3.3 Semantics

The formal semantics has an interpretation $\mathcal{I}$ which consists of the domain of interpretation ($\Delta^{\mathcal{I}}$) and an interpretation function $\cdot^{\mathcal{I}}$. The interpretation function maps every atomic concept A to the set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and every atomic role R to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

The interpretation of of the concepts are given on Table 3.2 on the following page.

$$
\begin{array}{rcl}
\top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} & = & \emptyset \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \\
(\geq nR.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid |\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}| \leq n\}
\end{array}
$$

Table 3.2: Interpretation of DL

C and D are concepts and an interpretation $\mathcal{I}$ is said to *satisfy* an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and an equality $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$. $\mathcal{I}$ satisfies the TBox $\mathcal{T}$ iff $\mathcal{I}$ satisfies every element of $\mathcal{T}$. If so it is said that $\mathcal{I}$ is a *model* of $\mathcal{T}$.

The ABox is the assertions regarding the individuals in the knowledge base. The interpretation function $\cdot^{\mathcal{I}}$ is expanded to map each individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. It is therefore important for the function to respect the *unique name assumption* (UNA). This means that if a and b are distinct names in the ABox then $a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Some examples of the semantics can be given if Mother and Woman are concepts and hasChild is an atomic role: $\exists$hasChild.Woman is the concept of people who have female children. $\forall$hasChild.Woman is the concept of people whose children are all women. The concept Mother$\sqcap\forall$hasChild.Woman would cover the Mothers who only have daughters. Finally the concept Mother$\sqcap \geq$ 3hasChild.$\top$ covers the concept of Mothers who has at least 3 children. $\top$ could be exchanged with Woman if the concept should cover mothers who has at least 3 female children.

3.4 Reasoner - RACER

DL offers reasoning services about explicit as well as implicit knowledge through inference. The basic inferences in DL are *satisfiability* and *subsumption*. A concept is satisfiable with respect to the TBox $\mathcal{T}$ if there exist a model $\mathcal{I}$ such that $C^{\mathcal{I}}$ is non-empty. A concept C is said to be subsumed by D if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for the model $\mathcal{I}$ of $\mathcal{T}$.

RACER [Haarslev and Möller, 2001] is a reasoner that can understand a wide range of DL of languages. It implements a number of optimization techniques that enables it to reason about TBox's and ABox's. RACER is a server program so it is used in conjunction with RICE (A RACER interactive client environment)[Möller et al., 2003] which enables navigation through a tree of concepts, view the ABox individuals and to make queries into the knowledge base.

RACER uses a Lisp-like concrete syntax known as Knowledge Representation System Specification (KRSS) for specifying axioms, defining assertions and for queries into the knowledge base. See Table 3.3 for the syntax used in this project syntax. See [Möller et al., 2004] for the complete syntax.

Name	Abstract syntax	Concrete syntax
Concept inclusion axiom	$C \sqsubseteq D$	`(implies C D)`
Concept assertion	$\mathsf{C}(a)$	`(instance a C)`
Role assertion	$\mathsf{R}(a,b)$	`(related a b R)`
Existensial quant.	$\exists\mathsf{R.C}$	`(some R C)`
Concept instances	C	`(concept-instances C)`
Role fillers	$\exists\mathsf{R}.I$	`(individual-fillers I R)`
Individual query	Checks if individual I is in the concept C	`(individual-instances?  I C)`

Table 3.3: Concrete syntax for RACER

Examples of the RACER syntax are:

```
(implies Woman Person)
(instance Mary Woman)
(related Mary Paul hasChild)
(individual-fillers Mary hasChild)
```

The examples creates a small knowledge base and requests the reasoner to query the knowledge base to find the children of Mary, to which it replies: Paul.

3.5 Software team

To model the software team in DL an TBox and ABox must be defined. The TBox and ABox will be modeled by the definition of the case study given in Chapter 2.4 on page 24. This also requires that SELinux is modeled in DL, but this will be done in Section 4.1 on the next page so this section seeks to identify notions from the case study that can be modeled without knowledge of the SELinux formulation of the case study.

One notion that is apparent to model in DL is the role hierarchy which can be built up using inclusion axioms:

$$\begin{aligned}
\mathsf{Supervisor} &\sqsubseteq \mathsf{Programmer}\\
\mathsf{Supervisor} &\sqsubseteq \mathsf{Tester}\\
\mathsf{Tester} &\sqsubseteq \mathsf{Member}\\
\mathsf{Programmer} &\sqsubseteq \mathsf{Member}
\end{aligned}$$

Other than that, further analysis must be made on the SELinux policy to further the construction of an DL knowledge base.

3.6 Summary

The chapter has introduced the concepts of DL and how a terminology can build the TBox and a how a set of assertions can specify the ABox. The case study was analyzed and the role hierarchy of it was modeled in DL. This was due to the fact that no other obvious concepts were identified and since the case study was already implemented in SELinux's security policy priority was focused on modeling that instead of coming up with a new way to represent the case study. The next chapter will show how the SELinux policy can be translated into DL.

CHAPTER 4

SELinux to Description Logic

This chapter provides a model formulation which can be used to model the relevant SELinux declarations as defined by the thesis objective. The model formulation is firstly explained whereafter translation rules for the different SELinux declarations are made to enable automatic translation of any SELinux policy. Finally, the procedure of translating the case study from SELinux to a DL knowledge base is explained.

The model formulation was inspired by the RBAC formulation in DL from [Chen Zhao and Lin, 2005].

4.1 Model formulation

An analysis of the different SELinux declarations must be made in order to specify the translation rules. Not all declarations are relevant in accordance to the objective of the thesis so some can be omitted. The audit declarations for instance are of little interest since the tool does not seek to model the running system. The declarations in Table 4.1 on the following page will not be translated. Note that not all declarations have been mentioned in Chapter 2 on page 9 for the same reason, their grammar can be found in Appendix A on page 83.

Declaration	Rationale
`neverallow`	The `neverallow` statements are used by SELinux's own policy parser `checkpolicy` [NSA, 2005] as assertions which must never be compromised by the policy. If an assertion fails, `checkpolicy` exists with an error and does not output the concatenated policy. Since it is a condition that the `checkpolicy` program exists with no errors, the `neverallow` assertion must be fulfilled and thus can be ignored.
`type_change`	Type change is used by SELinux-aware programs. Since the objective of this thesis is to model programs in the user domain (ordinary programs) `type_change` will be ignored.
`dontaudit`	Information about when not to audit interaction between two types is ignored, since the objective is not to model a running system.
`role_transition`	Role transition is typically used when restarting daemons. Daemon programs are not modeled are not the objective for this iteration of the tool.
`constrain`	Constrain declarations are used to restrict permissions further. The example policy specifies a few constraints which are there to restrict the ability to transition to different roles or to restrict users from certain domains.

Security contexts:
`sid`
`fs_use_xattr`
`fs_use_task`
`fs_use_trans`
`genfscon`
`portcon`
`netifcon`
`nodecon`
	The SIDs found in the policy are used by SELinux for system initialization and predefined objects. The remainder of security context declarations are concerned with the labeling decisions of the security server. Since it is not the objective to model the file system, these declarations are ignored.

Table 4.1: Ignored declarations

The remainder of the statements are to be modeled. To model SELinux as a knowledge base, it is clear that some general concepts are needed as well as atomic concepts. Furthermore, roles must also be defined which will be used to set up the wanted behavior of SELinux. Once general concepts and roles are defined, specific concepts and individuals will be added to model the case study.

There are some special cares that should be noted: The type allowance declaration builds a unique binding that defines the permissions when these two types interact. If the types interact with other types, the permission set could be different. The same applies to the permission and security class of the type allowance. This pair represents a unique permission on certain classes, that are different when the same class is paired up with another permission or vice versa.

To find out how to model the general declarations of the policy, basis was taken in a small example with connection to the case study, See Listing 4.1. The example is unique because it covers all the declarations that have been deemed useful in fulfilling the objective. And that it contains the macro expansion of the `domain_auto_trans` function Listing 2.1 on page 23 to ensure that the needed allow declarations are present to enable domain transition (type transition).

Listing 4.1: A simplified case study policy

```
 1  attribute domain;
 2  attribute file_type;
 3  type code_t, file_type;
 4  type tester_t, userdomain;
 5  type mosml_t, domain;
 6  type mosml_exec_t, file_type;
 7  allow mosml_t code_t:file read;
 8  allow userdomain mosml_t:process transition;
 9  allow userdomain mosml_exec_t:file { read { getattr execute } };
10  allow userdomain mosml_t:fd use;
11
12  allow mosml_t userdomain:fifo_file { ioctl read getattr lock write
        append };
13  allow mosml_t userdomain:process sigchld;
14  allow mosml_t userdomain:fd use;
15  allow mosml_t mosml_exec_t:file { read getattr lock execute ioctl
        };
16
17  allow mosml_t mosml_exec_t:file entrypoint;
18  type_transition userdomain mosml_exec_t:process mosml_t;
19
20  role tester_r types tester_t;
21  dominance{role supervisor_r {role tester_r;}}
22  allow supervisor_r tester_r;
23  user Tom roles supervisor_r;
24  user Alice roles tester_r;
```

It has been found useful to sketch concept-role relationships on a semantic graph, Figure 4.1 on the next page. This graph represents the way concepts and roles from the example are to be related. The oval objects indicates a role association while the boxes are square.

Note that Line 12 to Line 15 do not appear on the figure, to ease the readability.

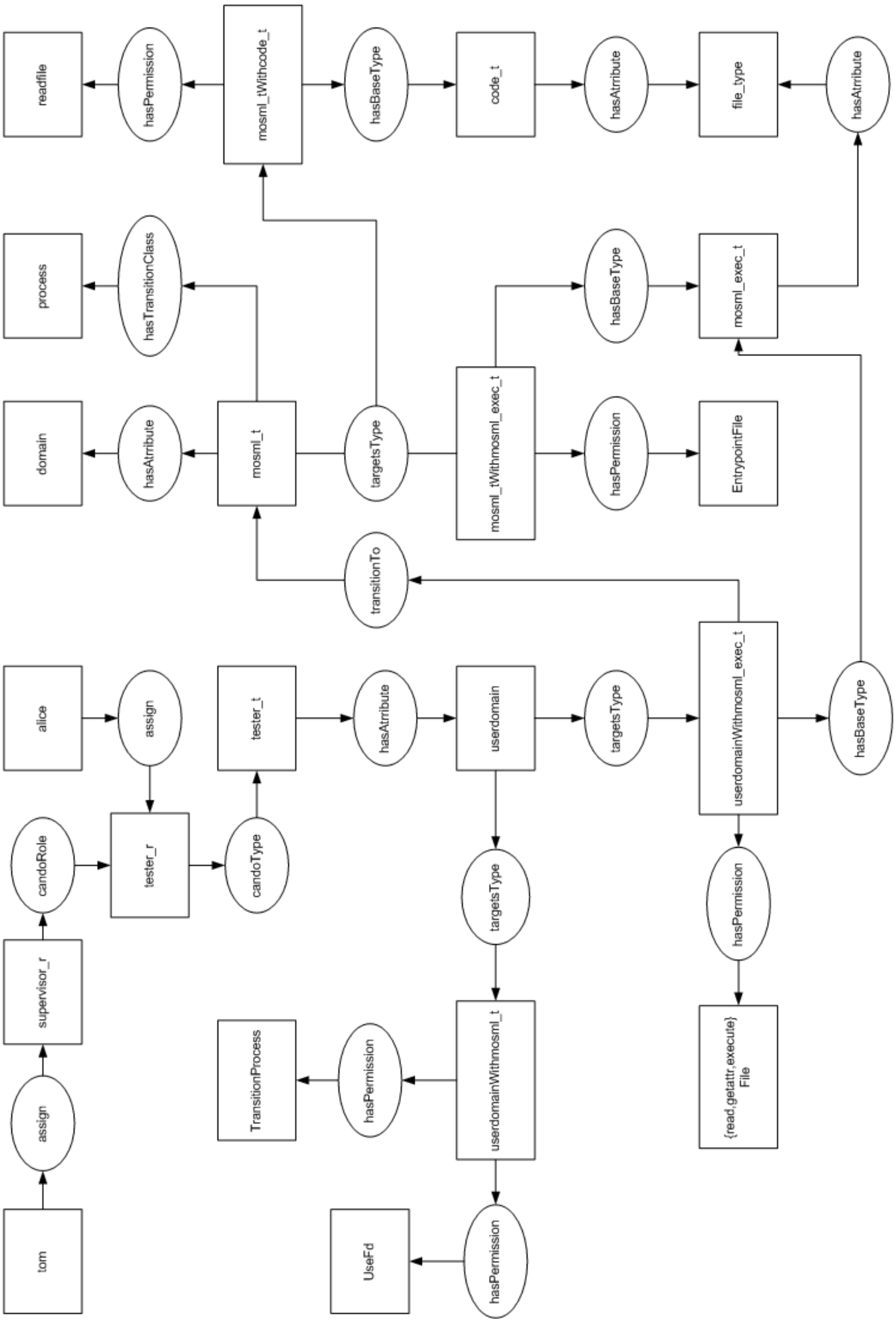


Figure 4.1: The Semantic Graph representing the policy of Listing 4.1 on the previous page

These lines would be incorporated into the graph in the same way the other type access nodes appear.

As it shall be seen in Chapter 6 on page 61 the semantic graph sketched here, has a similar counterpart in the actual model of the example.

Since an abstract view of what is to be modeled is in place, the time has come to define the alphabet of knowledge base $\mathcal{K}$. Notice that the sets referred to in this section were defined in Section 2.3 on page 13.

- User concept where all users reside.
- CRole concept of RBAC roles.
- Permission. All permissions are subsumed by this concept.
- Type is the types of the system.
- Class is the classes of permissions.
- Attribute is the concept of which all attributes are subsumed.
- for each role $r \in R_{\text{SELinux}}$, type $t \in T_{\text{SELinux}}$, class $c \in C_{\text{SELinux}}$ and attribute $a \in A_{\text{SELinux}}$ one atomic concept R, T, C, A respectively.
- for each permission $p \in PC_{\text{SELinux}}$ one P. Note that this is a permission paired with a security class.
- role assign binds a user to a role.
- role candoType binds a role to an allowed type.
- role targetsType binds two types together according to the allow statement.
- role hasPermission binds a type to a permission.
- role hasAttribute binds a type to an attribute.
- role hasBaseType binds a composite type with its base component.
- role transitionTo binds two types together for use with the type transition declaration.
- role hasTransitionClass binds a type and class together, to identify the class of the transition.
- role candoRole binds two roles to indicate the role allow rule.

The above alphabet enables the creation of a knowledge base where the case study can be modeled.

4.2 Translation rules

To use the above concepts and roles, some rules regarding the translation between the SELinux policy and DL must be defined.

The structure in the following sections resembles the structure from Section 2.3 on page 13 as they deal with the same syntax. The rules for defining the ABox are using a "*Tell and Ask*" [Levesque, 1984] interface, to put knowledge into the knowledge base. When adding knowledge into a concept an individual is needed for every concept. Since almost every concept in this model has a unique meaning it is only necessary to put one individual into the concept. The concepts in this model can be seen as singletons, the exception is the User concept which holds all the users of the system. The individuals in singletons are denoted by the subscript $_{\mathrm{I}}$. This means, for instance, that the concept readfile holds one individual $\{readfile_{\mathrm{I}}\}$, whereas the concept User can hold users $\{Tom, Alice, Bob, John\}$.

The rules are not coupled with examples, except for the first two types of declarations since these cover the procedure of translating the SELinux declarations.

4.2.1 TE declarations

Attributes

`attribute` *Attributename*;

The attribute declaration must be put into its super concept, and an individual of the attribute name is declared in the ABox.

TBox:
$\mathsf{Attributename} \sqsubseteq \mathsf{Attribute}$

ABox:
$\textsc{Tell}\{\mathsf{Attributename}(Attributename_{\mathrm{I}})\}$

Example:
`attribute domain;`
$\Rightarrow$

TBox:
$\mathsf{domain} \sqsubseteq \mathsf{Attribute}$

ABox:
$\textsc{Tell}\{\mathsf{domain}(\mathit{domain}_{\mathrm{I}})\}$

Type declaration

Let $\mathit{Attributes} \subseteq A_{\mathrm{SELinux}}$ then the translation of

`type` *Typename*, *Attributes*;

gives the following translation rules

TBox:
$\mathsf{Typename} \sqsubseteq \mathsf{Type}$
For each $A \in \mathit{Attributes}$: $\mathsf{Typename} \sqsubseteq \exists \mathsf{hasAttribute.A}$

ABox:
$\textsc{Tell}\{\mathsf{Typename}(\mathit{Typename}_{\mathrm{I}})\}$
For each $A \in \mathit{Attributes}$: $\textsc{Tell}\{\mathsf{hasAttribute}(\mathit{Typename}_{\mathrm{I}}, A_{\mathrm{I}})\}$

Example:
`type esales_t, domain;`
$\Rightarrow$

TBox:
$\mathsf{esales\_t} \sqsubseteq \mathsf{Type}$
$\mathsf{esales\_t} \sqsubseteq \exists \mathsf{hasAttribute.domain}$

ABox:
$\textsc{Tell}\{\mathsf{esales\_t}(\mathit{esales\_t}_{\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{hasAttribute}(\mathit{esales\_t}_{\mathrm{I}}, \mathit{domain}_{\mathrm{I}})\}$

Type attribute declaration

Let $\emptyset \subset \mathit{Attributes} \subseteq A_{\text{SELinux}}$ ($\mathit{Attributes}$ is a non-empty set of attributes) and
$\mathit{Typename} \in T_{\text{SELinux}}$ since $\mathit{Typename}$ has to be defined somewhere else.

Then

`typeattribute` *Typename Attributes*

becomes

TBox:
For each $A \in \mathit{Attributes}$: $\mathsf{Typename} \sqsubseteq \exists \mathsf{hasAttribute.A}$

ABox:
For each $A \in \mathit{Attributes}$: $\textsc{Tell}\{\mathsf{hasAttribute}(\mathit{Typename}_{\mathrm{I}}, A_{\mathrm{I}})\}$

The type attribute declaration adds more attributes to an already defined type. Since the type already have been put into the knowledge base, as have the attribute, the only thing that needs to be taken care of is adding the inclusion and assertion for the hasAttribute relation to both the TBox and ABox for each attribute.

Type access rules

Let $\emptyset \subset \mathit{Type}_S \subseteq D_{\text{SELinux}}$,
$\emptyset \subset \mathit{Type}_T \subseteq T_{\text{SELinux}}$,
$\emptyset \subset \mathit{Classes} \subseteq C_{\text{SELinux}}$ and
$\emptyset \subset \mathit{Perms} \subseteq PC_{\text{SELinux}}$ then the syntax

`allow|auditallow` $\mathit{Type}_S\,\mathit{Type}_T$: *Classes Perms*;

can be translated into the axioms:

TBox:
For each $T_S \in \mathit{Type}_S, T_T \in \mathit{Type}_T$:
$\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \mathsf{Type}$

$\mathsf{T}_S \sqsubseteq \exists \mathsf{targetsType.T}_S\mathsf{WithT}_T$
$\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \exists \mathsf{hasBaseType.T}_T$
For each $PermissionClass \in Perms$ $\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \exists \mathsf{hasPermission.PermissionClass}$

ABox:
For each $T_S \in Type_S, T_T \in Type_T$:
$\textsc{Tell}\{\mathsf{T}_S\mathsf{WithT}_T(T_S\,WithT_{T\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{targetsType}(T_{S\mathrm{I}}, T_S\,WithT_{T\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{hasBaseType}(T_S\,WithT_{T\mathrm{I}}, T_{T\mathrm{I}})\}$
For each $PermissionClass \in Perms$: $\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \exists \mathsf{hasPermission.PermissionClass}$

The `allow` and `auditallow` keywords can be treated alike for the same reason that `dontaudit` can be ignored, namely that audit information is not interesting for the purpose of this thesis. Note the differences of $Type_S$ and $Type_T$, namely that the former is a subset of domains, while the latter is a subset of types. Also note that the unique binding between source and target types are denoted by concatenating "With" between the two type names. This is similar to the unique permission and class binding, except the permission name and class name are concatenated without using a string in between the names.

Type transition

Let $\emptyset \subset Sourcetypes \subseteq T_{\mathrm{SELinux}}$,
$\emptyset \subset Targettypes \subseteq T_{\mathrm{SELinux}}$,
$\emptyset \subset Classes \subseteq C_{\mathrm{SELinux}}$ and
$Newtype \in T_{\mathrm{SELinux}}$ since $Newtype$ has to be defined somewhere:
The translation rule for

`type_transition` *Sourcetypes Targettypes:Classes Newtype*;

becomes

TBox:
For each $T_S \in Sourcetypes, T_T \in Targettypes, C \in Classes$,
$\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \mathsf{Type}$
$\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \exists \mathsf{hasBaseType.t}_T$
$\mathsf{Newtype} \sqsubseteq \exists \mathsf{hasTransitionClass.C}$
$\mathsf{T}_S\mathsf{WithT}_T \sqsubseteq \exists \mathsf{transitionTo.Newtype}$

ABox:
For each $T_S \in Sourcetypes, T_T \in Targettypes, C \in Classes$,
$\textsc{Tell}\{\mathsf{T}_S\mathsf{WithT}_T(T_S\,WithT_{T\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{hasBaseType}(T_S\,WithT_{T\mathrm{I}}, T_{T\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{hasTransitionClass}(Newtype_{\mathrm{I}}, C_{\mathrm{I}})\}$
$\textsc{Tell}\{\mathsf{transitionTo}(T_S\,WithT_{T\mathrm{I}}, Newtype_{\mathrm{I}})\}$

Type transition is fairly complicated since for every cartesian product of the types mentioned it must define the needed rules. The unique binding between source types and target types is defined as a new concept and included into the Type concept. The base type of the target type is related using the hasBaseType role and the actual transition is set by using the roles hasTransitionClass and transitionTo

4.2.2 RBAC declaration

Role declaration

Let $\emptyset \subset Types \subseteq D_{\mathrm{SELinux}}$ in

`role` *Rolename* `types` *Types*;

translates into

TBox:
$\mathsf{Rolename} \sqsubseteq \mathsf{CRole}$
For each $T \in Types$: $\mathsf{Rolename} \sqsubseteq \exists\mathsf{candoType.T}$

ABox:
$\textsc{Tell}\{\mathsf{Rolename}(Rolename_{\mathrm{I}})\}$
For each $T \in Types$: $\textsc{Tell}\{\mathsf{candoType}(Rolename_{\mathrm{I}}, T_{\mathrm{I}})\}$

Role allowance

Let $\emptyset \subset Currentroles \subseteq R_{\mathrm{SELinux}}$ and
$\emptyset \subset Newroles \subseteq R_{\mathrm{SELinux}}$ then becomes

TBox:
For each $R_C \in Currentroles$ and for each $R_N \in Newroles$:
$\mathsf{R}_C \sqsubseteq \exists\mathsf{candoRole}.\mathsf{R}_N$

ABox:
For each $R_C \in Currentroles$ and for each $R_N \in Newroles$:
$\textsc{Tell}\{\mathsf{candoRole}(R_{C\mathrm{I}},R_{N\mathrm{I}})\}$

Role dominance

Let $Dominatedroles \subseteq R_{\text{SELinux}}$ then

`dominance` {`role` *Headrole* {`role`; *Dominatedroles*}}

translates to

TBox:
$\mathsf{Headrole} \sqsubseteq \mathsf{CRole}$
For every dominated role $R_D \in Currentroles$: $\mathsf{Dominatedroles} \sqsubseteq \mathsf{Headrole}$

ABox:
$\textsc{Tell}\{\mathsf{Headrole}(Headrole_{\mathrm{I}})\}$

4.2.3 Users

User declaration

Let $\emptyset \subset Roles \subseteq R_{\text{SELinux}}$ in

`user` *UserID* `roles` *Roles*;

becomes **ABox:**
$\textsc{Tell}\{\mathsf{User}(UserID)\}$
For every role $R \in Roles$: $\textsc{Tell}\{\mathsf{assign}(UserID,R_I)\}$

4.3 Software team

This section will highlight a simplified version of the software team definition. It will also provide motivation for the automatic generated DL knowledge base.

To showcase the translation rules a few declarations were taken out from the software team's policy from Chapter 2, Section 2.4 on page 24. The selected policy snippet can be seen on Table 4.2. The selected policy declarations feature the objective of getting the user `Bob` authorized for the tester role and authorizing that role to for the `mosml_` domain and granting read access to `code_t` files.

1. `user Bob roles tester_r;`
2. `role tester_r types mosml_t;`
3. `allow mosml_t code_t:file read;`

Table 4.2: Policy snippet

Currently, the following axioms are declared when processing the above three declarations. The ABox definition follows the same pattern and will be omitted here for declaration #2 and #3.

1. $\textsc{Tell}\{\mathsf{User}(Bob)\}$
1. $\textsc{Tell}\{\mathsf{assign}(Bob, tester\_r_I)\}$
2. $\mathsf{tester\_r} \sqsubseteq \mathsf{CRole}$
2. $\mathsf{tester\_r} \sqsubseteq \exists \mathsf{candoType.mosml\_t}$
3. $\mathsf{mosml\_tWithcode\_t} \sqsubseteq \mathsf{Type}$
3. $\mathsf{mosml\_t} \sqsubseteq \exists \mathsf{targetsType.mosml\_tWithcode\_t}$
3. $\mathsf{mosml\_tWithcode\_t} \sqsubseteq \exists \mathsf{hasBaseType.code\_t}$
3. $\mathsf{mosml\_tWithcode\_t} \sqsubseteq \exists \mathsf{hasPermission.readfile}$

Table 4.3: Policy snippet translated to DL

4.4 Alternative model formulations

Alternative model formulations were considered. The SELinux declaration that raises the most issues is the type access declaration. The declaration can be seen as a chain of knowledge that must be modeled in the knowledge base, see Figure 4.1 on page 40. An option is to have more implicit knowledge attributed to the concept name than currently (i.e. aWithb concept names for types a and b and the permission concepts which is a concatenation of permission and class). This would imply fewer concepts but more roles.

Below are the resulting axioms when applying the alternative model formulation for translating the policy from Table 4.2 on the facing page. Note that ABoxes are still only shown for the user declaration, as they are analogues to the TBox axioms.

1. $\textsc{Tell}\{\mathsf{User}(\mathit{Bob})\}$

1. $\textsc{Tell}\{\mathsf{associatedTo}(\mathit{Bob},\mathit{tester\_r\_mosml\_t\_code\_t\_file\_read}_I)\}$

2 and 3. $\mathsf{tester\_r\_mosml\_t\_code\_t\_file\_read} \sqsubseteq \mathsf{ComplexType}$

The alternative model formulation was not used since such a high degree of implicit knowledge attributed to the concept names is unwanted. Furthermore, (DL) roles would be needed to support a role hierarchy, specify the permission class, source and target types etc.

The option to have a bit less knowledge to the concept names could form yet another alternative model formulation, which can be seen used below:

1. $\textsc{Tell}\{\mathsf{User}(\mathit{Bob})\}$

1. $\textsc{Tell}\{\mathsf{assign}(\mathit{Bob},\mathit{tester\_r}_I)\}$

2. and 3. $\mathsf{tester\_r} \sqsubseteq \exists \mathsf{associatedTo}.\mathsf{mosml\_t\_code\_t\_file\_read}$

2. and 3. $\mathsf{mosml\_t\_code\_t\_file\_read} \sqsubseteq \mathsf{ComplexType}$

Such a formulation has less implicit knowledge associated to the concept names. The process of identifying a chain of knowledge as mentioned above and decomposing it into smaller pieces can be continued until the used model formulation from Section 4.1 on page 37 is (re)found. That model also have some implicit knowledge in the concept names (i.e. aWithb for types a and b and permission pc for permission p and class c), but such concepts are necessary in order to keep the unique binding between the types or permissions and classes.

The reason that these simpler alternate model formulations are unwanted is that they add a large amount of constraints to the syntax of the concept names. The concept names would have to follow a recognizable pattern, if any automated queries regarding the model are to be created, see Section 6.3 on page 65.

4.5 Summary

This chapter presented a model formulation that enabled the SELinux policy to be modeled in DL. After the formulation was in place, translation rules for modeling the various declarations of SELinux's security policy was presented. The translation rules does not cover all of the declarations of SELinux, but it covers a range such that the objective of the thesis has been fulfilled. The translation rules were applied to a small set of the case study to showcase the rules and alternative formulations were discussed.

The chapter shows that the security policy systematically can be translated which leads to a wish for an automated process that does exactly that. An implementation of the translation rules is particulary useful since the security policy is very large. Even for small examples the translation rules specifies many DL axioms, which further motivates an automated process. The following chapter illustrates how such implementation can be achieved.

Chapter 5

Implementation

This section presents the implementation design and an overview of the structure of an automated tool that translates an SELinux policy into a DL representation of it. The chapter shows the design goals and the steps needed in order to complete these goals. The chapter is concluded by sketching a test strategy for the automated tool.

5.1 Design

The design of the tool is based on solving two goals.

- Inputting the SELinux policy into the tool.
- Using the translation rules found in Section 4.2 on page 42 to output a DL knowledge base.

The first objective is to build a Scanner/Parser that can translate the SELinux's policy into an abstract data structure which can be used to solve the next goal. When that is done, the tool can process the data structure and automatically generate the knowledge base.

Standard ML was chosen as the implementation language, specifically Moscow ML [Romanenko et al., 2005] which is a lightweight implementation of Standard ML. ML is a functional language with type checking and type inference which is beneficial for data modeling. Moreover, ML in conjunction with Lex[Lesk, 1975] and Yacc[Johnson, 1979] has excellent abilities to define rules for scanning and producing the parsing capabilities needed to input the SELinux policy file.

The structure of the parsing files (`parse.sml`, `Lexer.lex` and `Gram.grm`) are based on inspiration from the Moscow ML distribution examples.

5.2 Lexical analysis

To use Lex it is required to identify the keywords for use of the lexical translation of keywords to tokens. A small section of the process will be presented here. All the different keywords can be found in the grammar in the appendix and the final Lex specification can be found in Appendix B.2 on page 91.

Analyzing the grammar, tokens are found and defined for Lex. The tokens are:

- Keywords such as `"allow", "role", "nodecon", "not", "if"` etc.
- Symbols such as `"*", "-","==", ";"` etc.
- Numbers
- Paths
- IPv6 addresses

Before specifying the grammar of the SELinux policy for Yacc, it is necessary to define an abstract data model that Yacc will use for the resulting parse tree.

5.3 Abstract data model

The abstract data model can be found in Appendix B.1 on page 89. It is clear that the abstract data model is based on those declarations that have been chosen for translation in the Translation rule section. The abstract data model shares much of the structure from SELinux's grammar. It differs from SELinux's grammar to facilitate a simpler design of the parse tree.

```
datatype Decl =
      Common_perms_def of string * string list
    | Class_def of string
    | Class_def_perms of string * string list
    | Class_def_inherit  of string * string
    | Class_def_inherit_perms of string * string * string list
    | Attrib_decl of string
    | Type_decl of string * string list * string list
    | Typealias_decl of string * string list
    | Typeattribute_decl of string * string list
    | Type_transition_rule of Set * Set * Set * string
    | AllowDecl of Set * Set * Set * Set
    | AuditAllowDecl of Set * Set * Set * Set
    | Role_decl of string * Set
    | Role_dominance of Roles
    | Role_allow_rule of Set * Set
    | Role_transition of Set * Set * string
    | User_decl of string * Set
    | If of Cond_expr *  Block * Block
    | Bool_def of string * Bool_val
    | SKIP
withtype Block = Decl list;
```

The abstract data model follows the same concrete syntax structure as specified in Section 2.3 on page 13 such that the when the concrete syntax of type access, for instance, is `allow Ts Tt:C P;` the abstract data model's data constructor `AllowDecl of Set * Set * Set * Set` is `AllowDecl(Ts, Tt, C, P)`. This holds for the other types of declarations also.

The policy is modeled as a list of declarations. The main datatype is the `Decl` datatype which holds the type of declarations that can be found in the policy. The `Decl` datatype uses a few other datatypes to model the declarations. Most noteworthy are the `Set` datatype and the `Roles` datatype which also uses other datatypes. The datatypes are the result of analysis of the grammar. One special constructer in the `Decl` datatype is the `SKIP` constructor. `SKIP` is the no operation constructor of the interpreter. It is used whenever the parser meets a declaration that is eligible to be ignored c.f. Table 4.1 on page 38.

5.4 Parsing

After the abstract data model has been defined, the time has come to define the grammar for use of Yacc. The grammar used by Yacc is also based on SELinux's grammar from the Appendix. The datatypes from the abstract data model are used in relation with the grammar to specify the nodes in the resulting parse tree.

```
Decl :
.
.
.
  | ALLOW Set Set COLON Set Set SEMI    { AllowDecl($2,$3,$5,$6) }
  | ROLE IDENTIFIER TYPES Set SEMI      { Role_decl($2, $4) }
.
.
.
  | ROLE_TRANSITION Set Set IDENTIFIER SEMI    { SKIP }
  | SID IDENTIFIER                             { SKIP }
.
.
.
;
```

Table 5.1: Grammar snippet

A snippet of the grammar specification can be seen on Table 5.1. It shows the relation with the abstract data model and the grammar. The grammar shows how the Allow declaration (line 5) is built up(left side), and how the parser should create the parse tree in accordance with the abstract data model, which was `AllowDecl of Set * Set * Set * Set` (right side). Note that even though declarations are to be ignored the grammar must still be defined and correctly formed in order for the parser to accept the complete policy file. The policy file is required to be well-formed (c.f. the conditions set in Section 1.2 on page 4) but the parser will perform (a superfluous) syntax check to see if the condition still holds.

Using Lex and Yacc generates the parser that will scan the SELinux policy in accordance to the grammar. Running the parser completes the first goal, namely converting the "raw" policy file into an abstract data model.

5.5 Translation

The next goal is to traverse the parse tree and translate the declarations into a DL knowledge base. As mentioned in Section 2.3.1 on page 15, forward reference to type names is allowed, so this must be handled. Furthermore, the `Set` constructor supports the `*` (asterisk) symbol, which means whenever the asterisk symbol appears in a set, the set is represented by the universe of whatever the set represents. As specified in the grammar and noted in Section 2.3.7 on page 23 the `Set` constructor can be used by represent, types, classes, permissions and roles in the system.

Finally, the conditional declaration found in the SELinux policy also allow forward references to the booleans used. So the tool needs to collect the booleans the first, and then use them to evaluate the condition.

The above indicates that the parser must traverse the parse tree two times. Upon the first traversal the data declared by declarations found on Table 5.2 on the following page will be collected.

The used types are modeled as follows:

```
type StringSet = string Binaryset.set;

type CommonMap = (string, StringSet) hash_table;
type BoolMap = (string, bool) hash_table;
type AttributeMap = (string, StringSet) hash_table;
type AliasMap = (string, StringSet) hash_table;
type PermissionMap = (string, StringSet) hash_table;
type RoleMap = (string, string list) hash_table;
type ClassSet = StringSet;
type AttributeSet = StringSet;
type TypeSet = StringSet;
type RoleSet = StringSet;
type UserSet= StringSet;
```

From the type model, it shows that there are two basic kinds of complex types, namely the string set, denoted by the suffix Set and some kind of table, denoted by the suffix Map.

Once the types have been modeled, the function which performs the first pass can be modeled as follows:

```
PT1: parsetree * ClassSet * PermissionMap * BoolMap * CommonMap *
                 AttributeSet  * TypeSet * AliasMap, AttributeMap *
                 RoleSet * UserSet
->
                 ClassSet * PermissionMap * BoolMap * CommonMap *
                 AttributeSet * TypeSet * AliasMap * AttributeMap *
                 RoleSet * UserSet
```

The input types are initially empty. The function recursively traverses through the parsetree collecting data and putting it into the appropriate data structures according to the type of declaration. Upon termination, the output types now contain data representing the data found in the parsetree.

The function is defined by pattern matching and will match the following declarations which are related to data declaration and ignore (skip) everything else.

- `Common_perms_def (name,list)`
- `Class_def name`
- `Class_def_perms (name,list)`
- `Class_def_inherit (name,inherit)`
- `Class_def_inherit_perms (name, inherit,list)`
- `Attrib_decl name`
- `Type_decl(name, aliases, attributeNames)`
- `Typeattribute_decl (name, attributeNames)`
- `Typealias_decl (name, aliases)`
- `Role_decl (name, _)`
- `User_decl(name, _)`
- `Bool_def(name, b)`

Table 5.2: The declarations matched on the first traversal of the parsetree

After the first pass though the parsetree, all the data needed to process the policy's fundamental access declarations has been gathered. The next pass can process the remainder of the declarations and use the translation rules to create the necessary DL declarations that defines the knowledge base.

The function that runs through the parse tree the second time has a similar signature to the first one.

```
PT2: parsetree * ClassSet * TypeSet * AliasMap * RoleSet *
                 PermissionMap * AttributeMap * BoolMap * "KB written"
->
                 ClassSet * TypeSet * AliasMap * RoleSet *
                 PermissionMap * AttributeMap * BoolMap * WithSet
```

PT2 differs in signature from PT1 with CommonMap, UserSet and WithSet. PT2 does not need information regarding users or the common constructor, as they do not appear in a Set construction. The function outputs a new set, WithSet which is the result of the translation rules regarding type access and type transition. It is necessary to create this new set and not add it to the existing TypeSet since WithSet is not in the universe of types regarding the Set constructor in the grammar.

The PT2 function is also a pattern matching function. The matched declarations are seen on Table 5.3. The second traversal deals with the definition of the knowledge base according the translation rules. The function uses the translation rules as specified in Section 4.2 on page 42 to write to a file for every declaration it meets, except the If declaration.

- `If(expr, b1, b2)`
- `Attrib_decl name`
- `Type_decl(name, aliases , attr)`
- `Typeattribute_decl(name, attr)`
- `AllowDecl(Ts, Tt, C, P)`
- `AuditAllowDecl(Ts, Tt, C, P)`
- `Type_transition_rule(Ts, Tt, C, name)`
- `Role_decl (name, AllowedTypeSet)`
- `Role_dominance roles`
- `Role_allow_rule (Rs, Rt)`
- `User_decl(name, roles)`

Table 5.3: The declarations matched on the second traversal of the parsetree

The If declaration does not have a translation rule. This constructor is used by the SELinux's policy to condition some statements. The PT2 function evaluates the conditional expression by looking up the booleans used and then selecting the block which evaluates to true. Note that the conditional declaration only allows type allowance and type transition declarations, but the grammar specifies that a block is a `Decl list`. Strictly, this is wrong as it allows for all kinds of declarations to appear in a block, and thus requiring another pass of the parsetree, but since the conditions of the thesis state that the policy conforms to a format of which SELinux's own parser accepts, it must be assumed that only the allowed declarations appear within a block.

After traversing through the tree, two function calls are necessary: The first is `finalizeKB: ClassSet * PermMap -> "KB written"` which produces the concept inclusions and assertions of for every $pc \in PC_{\text{SELinux}}$ and $c \in C_{\text{SELinux}}$. This step is done at this point to optimize the performance since if it was done whenever a type allowance declaration was met, would introduce many redundant concepts. The function does not output anything but appends to the files which `PT2` started on.

Running `PT1`, `PT2` and `finalizeKB` specifies the TBox and ABox, but RACER needs to know the set up of the knowledge base so the second function which needs to be run after the parse tree has been traversed is `createSignature` which is modeled as

```
type sigName = string
createSignature: string * ClassSet * TypeSet * RoleSet * PermMap
                          * AttSet * UserSet * WithSet)
-> "Signature written"
```

The function manipulates a file that specifies for RACER the different classes, types, (SELinux) roles, permissions, attributes and users as well as the (DL) roles individuals and concepts there exists in the knowledge base.

5.6 Testing strategy

The testing strategy is also divided according to the implementation goals. First, testing should be done on the Scanner and Parser. Second, testing the translation functions must also be done.

Testing the Scanner/Parser must also be done in accordance to the overall objective, namely that the tool should be able to successfully parse the example policy that SELinux's `checkpolicy` program has created. This means that if the Scanner/Parser can produce a parsetree based on the policy file and the parsetree corresponds to the policy file's constructors, the Scanner/Parser has fulfilled its goal. Moreover, introducing syntax errors in the policy file must stop the Scanner/Parser.

To test the translation, the different kinds of declarations should be matched up with the translation rules and checked to see that they correspond. Care should be taken to verify that the abstract data model's constructors are followed such that the resulting DL specification is complete with regard to the translation rules.

5.7 Summary

An implementation design for automated generation of a knowledge base was presented and executed in Moscow ML using Lex and Yacc as lexer and parser generators. The implementation followed the translation rules as specified in Section 4.2 on page 42. A testing strategy was created but was not systematically followed due to lack of time. During the implementation the testing strategy was informally followed, i.e. the various functions were informally tested after each component finished implementation. The structure of the parsetree was analyzed to verify the structure matched that of the policy file. The functions and types found in Appendix B.7 on page 102 were used to for the informal tests.

The following chapter sets out to investigate whether the translated policy corresponds to the declarations from the policy. Furthermore, queries are written to illustrate how to use the reasoner.

CHAPTER 6

Verification

This chapter takes the model formulation from Chapter 4 on page 37 into use, but it also attempts to determine if the knowledge base created is useful to such a degree that the knowledge base can be said to model the policy defined in SELinux's security policy. After the verification has taken place, a catalog of queries are created to showcase some of the useful queries that can be made regarding the model of SELinux. Finally, queries are made regarding a small scale of the case study.

6.1 Verification Strategy

To verify that the modeled knowledge base models the SELinux's policy, some goals of verification is needed. The method of verification is discussed and a set of general queries are defined and the RACER queries that extracts the wanted information is described.

The method of verification is to test the different atomic concepts and roles to see if they yield the expected information. This means that the different axioms are verified on a small scale and will then be scaled up to the complete knowledge base. This strategy is feasible since every axiom adds to the knowledge base such that the knowledge base is complete at every given time. An intermediate knowledge base does not *model* the SELinux policy of course, but models a subset of it until the final axiom regarding it has been added to the knowledge base.

6.2 Verification

The main focus of the verification is to test the axioms where roles are used such as $\mathsf{Typename} \sqsubseteq \exists \mathsf{hasAttribute.A}$. Simple axioms such as $\mathsf{Typename} \sqsubseteq \mathsf{Type}$ are only verified for one example and is then assumed to behave similarly with others axioms.

The simplest way to verify the model is to set up a small example and verify that the different axioms behave as expected. It is important that the example invokes all the different DL roles, but not necessarily all the different SELinux syntaxes since this is a verification of the model and not of the translation rules. The example policy from Listing 6.1 accomplishes the desired goal and is supposed to represent a small subset of the case study.

Listing 6.1: A very small SELinux example policy

```
 1  attribute domain;
 2  type code_t;
 3  type tester_t, domain;
 4  type mosml_t, domain;
 5  type mosml_exec_t;
 6  allow mosml_t code_t:file read;
 7  type_transition tester_t mosml_exec_t:process mosml_t;
 8  allow tester_t mosml_exec_t:file {read getattr execute};
 9  allow mosml_t mosml_exec_t:file entrypoint;
10
11  role tester_r types tester_t;
12  dominance{role supervisor_r {role tester_r;}}
13  allow supervisor_r tester_r;
14  user Tom roles supervisor_r;
15  user Alice roles tester_r;
```

The verification can in part occur visually using RICE's ability to list individuals in the concepts, but also due to the fact that RICE can create a graph of the (DL)role assertions made by the ABox. The role assertion graph can be seen on Figure 6.1 on page 64. *TESTER_TWITHMOSML_EXEC_T__I* has two hasbaseType role assertions to *MOSML_EXEC_T__I*. This is because of the translation rules (see Section 4.2.1 on page 42) of Line 7 and Line 8. It introduces no conflict in the DL model because of the UNA (unique name assumption), the model sees it as one assertion.

Note the resemblance with Figure 6.1 on the next page and Figure 4.1 on page 40. The difference is that Figure 4.1 has a few more permissions due to the policy upon it was based on and that it contains a userdomain concept. The userdomain concept appears because the figure was based on an unprocessed policy file. A processed policy file would have substituted the `userdomain` attribute with all the types which it described, in this case the tester_r. This means that the role associating the userdomain with the tester_t type would disappear and the role associations from the userdomain would be substituted with the tester_t type instead, exactly as seen on Figure 6.1 on the next page. This does not verify the model, but it shows a good resemblance from what the translation rules initially set out to model and the actual model. To verify the model formally, it should be proved that the knowledge base models what is in the SELinux policy and nothing else, meaning that it has to be shown that the declaration interpretation appears in the model, but also that the model does not expand upon what can be found in the policy.

RICE is also able to create a graph of the TBox. Since the TBox is a very flat structure, only a part of the graph can be seen on Figure 6.2 on page 65. Note the role hierarchy and that domain is below Attributes.

With the Figure 6.3 on page 66 the simple concept axioms and the ABox have been verified. The concept axioms are shown as a hierarchical structure on Figure 6.2 on page 65. The ABox's role assertions are shown on Figure 6.1 on the next page and as can be seen on Figure 6.3 on page 66, RICE can be used to verify the contents of the concepts which is what the concept assertions handle. The concept content is listed to the right of the figure below `ABoxes` and `Instances`.

What is left to be verified are the concept axioms using role fillers. There are 9 in total: assign, candoType, targetsType, hasPermision, hasAttribute, hasBaseType, transitionTo, hasTransitionClass and candoRole. The procedure for testing each role is the same, use the RACER query `(concept-instances (some R C))` to test role R with regard to some concept C where a known relation should exist. So, for the assign role, the query to determine that Line 11 from Listing 6.1 on the preceding page has been successfully modeled is:

```
(concept-instances (some assign tester_r))
```

Which replies: `(TOM ALICE)`. This means that Tom and Alice are bound to the tester_r concept. The reason that they both appear is because of the role hierarchy specified by Line 12, also from Listing 6.1 on the facing page (The remaining line references in this section all refer to Listing 6.1 on the preceding page). For the other roles, there are no implicit knowledge, so for

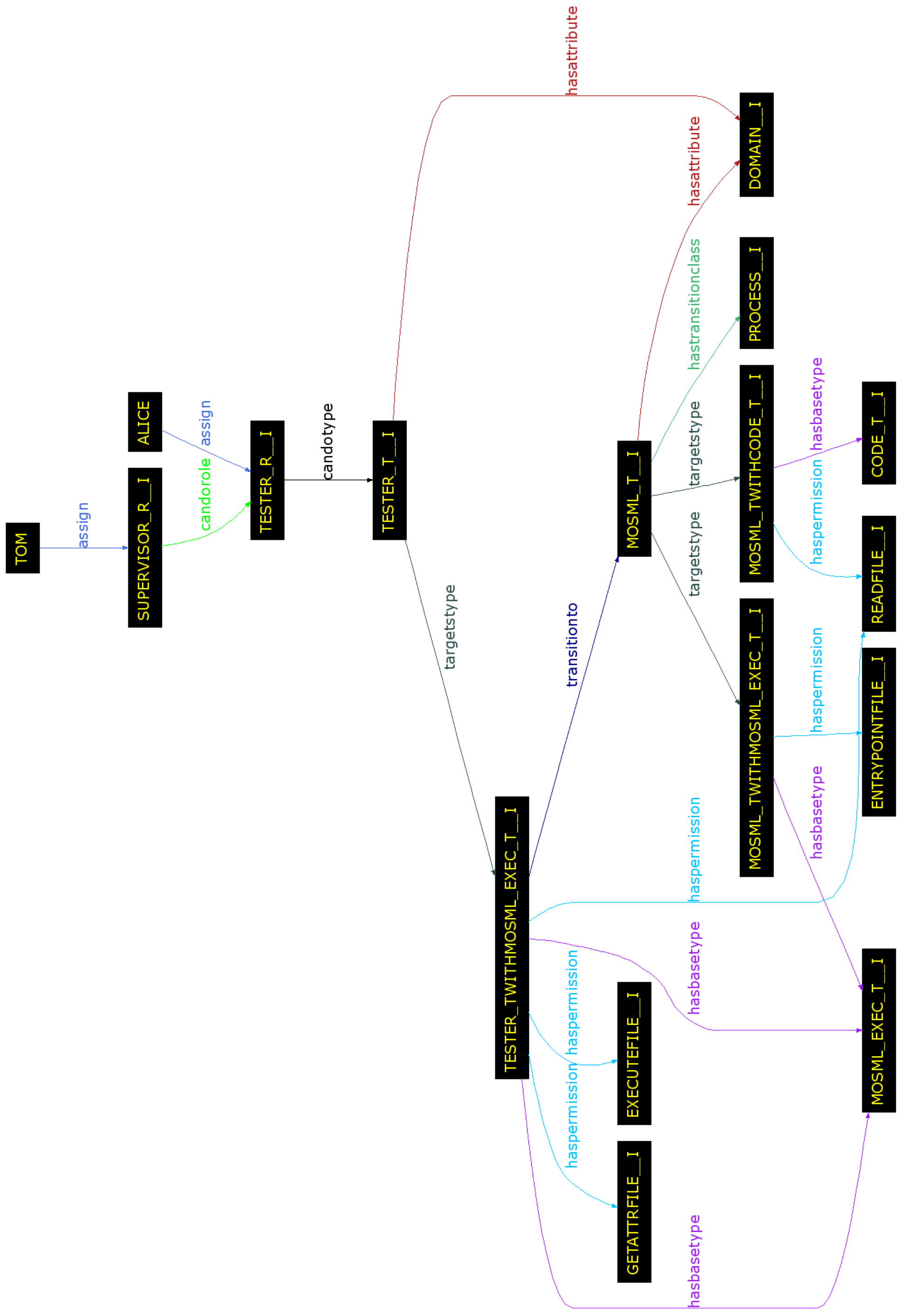


Figure 6.1: The role assertion graph of the example policy

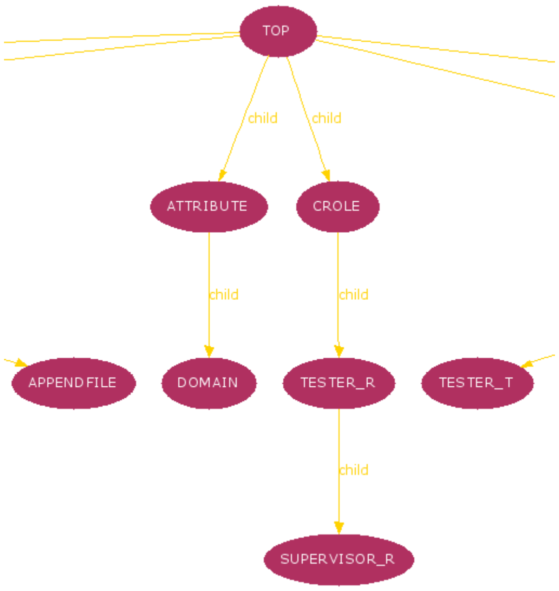


Figure 6.2: The concept assertion graph of the example policy

`(concept-instances (some hasAttribute domain))` the reply is

`(MOSML_T__I TESTER_T__I)`

which is specified explicitly by Line 3 and Line 4.

This procedure can be repeated for the remaining seven roles with similar results. The complete listing can be seen in Table 6.1 on page 67.

Having done that, it is assumed that the knowledge base models the SELinux example defined by Listing 6.1 on page 62 and moreover it is assumed that any knowledge base translated by the rules set forth in Chapter 4 on page 37 models the SELinux policy under the conditions defined by the objective in Section 1.2 on page 4.

6.3 Queries

A knowledge base in itself is not that interesting. To make the knowledge base useful, a catalog of queries have been developed to extract knowledge from the

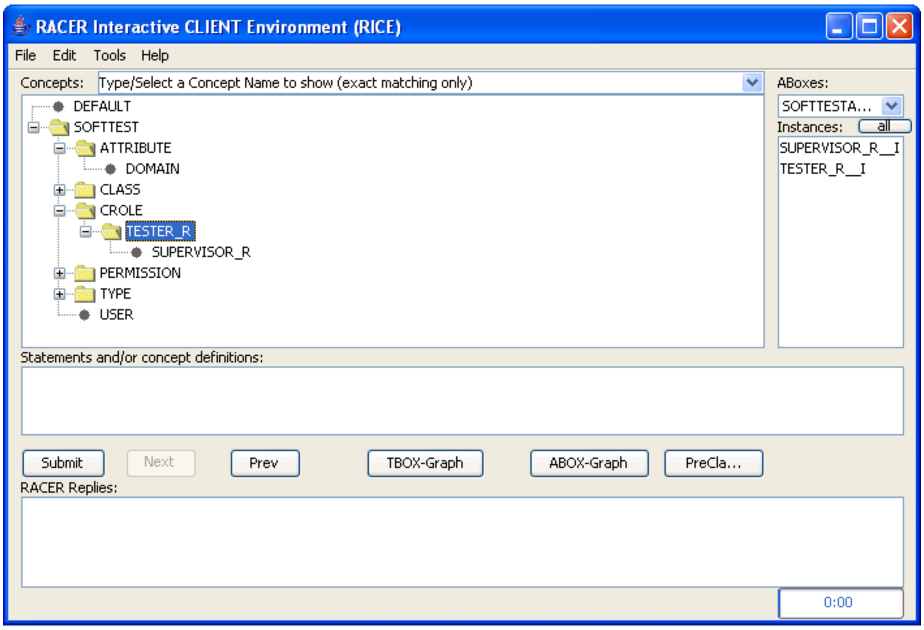


Figure 6.3: The concept assertion graph of the example policy

model. The queries are first defined as general questions and are afterwards specified as a set of queries that aims to answer the following questions. The questions are relevant in the sense that it is interesting to formulate a high level question and then get told which queries to run. Such an behavior could be automated. Note that some queries uses individuals, while other use concepts. The a human user is able to utilize the fact that the concepts are singletons and the individual residing in it is the same name with `__I` appended. An automated process would benefit from having no such assumption, so note that the query `(individual-direct-types In)` can be used to specify the individual *In* concept and the query `(concept-instances C)` can be used to specify the individuals residing in concept `C`.

1. Recall that an attribute represents a set of types. Which types are associated with the attribute A?
2. Who is assigned to role R?
3. Which permissions does program P have?
4. What permissions does role R have?
5. Is user U allowed to run program P?

RACER query	RACER reply
`(concept-instances (some assign tester_r))`	`(ALICE TOM)`
`(concept-instances (some candoType tester_t))`	`(SUPERVISOR_R__I TESTER_R__I)`
`(concept-instances (some targetsType mosml_tWithcode_t))`	`(MOSML_T__I)`
`(concept-instances (some hasPermission executefile))`	`(TESTER_TWITHMOSML_EXEC_T__I)`
`(concept-instances (some hasAttribute domain))`	`(MOSML_T__I TESTER_T__I)`
`(concept-instances (some hasBaseType code_t))`	`(MOSML_TWITHCODE_T__I)`
`(concept-instances (some transitionTo mosml_t))`	`(TESTER_TWITHMOSML_EXEC_T__I)`
`(concept-instances (some hasTransitionClass process))`	`(MOSML_T__I)`
`(concept-instances (some candoRole tester_r))`	`(SUPERVISOR_R__I)`

Table 6.1: Verification of concept axioms involving roles

This set of queries can easily be expanded so it should not be seen as the set of interesting queries but merely a subset which showcases the usability of having the policy modeled in DL. Some questions have already been answered in the verification phase, e.g. the `(concept-instances (some hasAttribute domain))` query answers Question 1 on the facing page with regard to the domain attribute and
`(concept-instances (some assign Programmer_r))` answers the Question 2 with regard to programmers. Question 3, 4 and 5 involves more queries than one, so they will be showed by using queries upon the example from Listing 6.1 on page 62.

Question 3 is nice to answer first since the queries formulated can be used in relation to the other questions. First, though, it is needed to specify the question further since programs in SELinux can have different permissions depending on the user (and role) that started it. So it is beneficial to revise the question to: Which permissions does domain P have? Often only one domain is defined per program, but the distinction is needed in case a type transition specifies a different domain upon executing the same file. So, for the question regarding the program domain of `mosml` the concept mosml_t is considered. The queries can be found on Table 6.2 on the following page.

RACER query	RACER reply
`(individual-fillers mosml_t__i targetsType)`	`(MOSML_TWITHCODE_T__I MOSML_TWITHMOSML_EXEC_T__I)`
`(individual-fillers MOSML_TWITHCODE_T__I hasPermission)`	`(READFILE__I)`
`(individual-fillers MOSML_TWITHMOSML_EXEC_T__I hasPermission)`	`(ENTRYPOINTFILE__I)`

Table 6.2: Query example of determining the domain permissions for mosml_t

Question 4 can be answered by first determining what types (domains) the role are authorized to and then finding out what the permissions are for those domains. The queries used to find the authorized types can be found on Table 6.3 and the result can be used with the queries that answers Question 3 to determine which permissions are attributed to that domain.

RACER query	RACER reply
`(individual-fillers tester_r__i candoType)`	`(TESTER_T__I)`

Table 6.3: Query example of the types authorized for role tester_r

Answering Question 5 is fairly complex. It involves a few "jumps" back and forth in the (DL) role assertion graph. First of all, the question is rephrased to ask if the user U is allowed to enter the program domain P. The question will be answered by an example. The question is Tom allowed to enter the `mosml_t` domain. The queries used are found on Table 6.4 on the facing page

6.4 Software team

The case study was implemented into an SELinux policy as defined in Section 2.4 on page 24. If the translation rules were run on this example it would generate so many axioms that the reasoner was not able to answer any queries regarding the system. Because of the explosion of axioms, it was decided to focus on the intended behavior and not use the macro `full_user_role`. Furthermore, any declarations with no relation to the case study were removed. Even so this reduced policy consisted of roughly 1,500 declarations which was translated into approximately 19,000 axioms to represent the knowledge base (The raw data for the TBox and ABox files were approximately 1MB in size). As a comparison, the `full_user_role` macro applied to the policy in which the case

RACER query	**RACER reply**
First of all it must be determined if the domain in question can be accessed at all, if not, the answer is no and the query is answerd.	
`(individuals-related? mosml_t__i process__i hasTransitionClass)`	`T`
If yes (`"T"`) find out which type bindings has access to the domain, see Table 6.1 on page 67 regarding the transitionTo query.	
`(concept-instances (some transitionTo mosml_t))`	`(TESTER_TWITHMOSML_EXEC_T__I)`
To find out if Tom has been granted access to the found type, it is needed to invistigate his role.	
`(individual-fillers tom assign)`	`(SUPERVISOR_R__I)`
Using the queries for Question 4 it is found out that a supervisor has no types authorized. It is then needed to investigate if that role dominates any other roles	
`(concept-parents supervisor_r)`	`((TESTER_R))`
It is then needed to determine which types the role is authorized for. `(individual-fillers tester_r__i candoType)`	`(TESTER_T__I)`
The next query tests to find out if there is a link between the type that has access to the `mosml_t` domain and the user Tom `(individual-fillers tester_t__i targetsType)`	`(TESTER_TWITHMOSML_EXEC_T__I)`
It has been found that Tom is authorized to a role, which has access to a type, that can interact with another type, which can transition to the `mosml_t` domain. It is still needed to check if such a transition is allowed.	
`(individual-instance? TESTER_TWITHMOSML_EXEC_T__I (some hasPermission executefile))`	`T`

Table 6.4: Query example of determining whether a user can execute a program

study was implemented creates around 20,000 SELinux declarations and the complete knowledge base when modeling this was 700MB in size.

Question 5 on page 66 is sought answered in the software team for user and for the `nedit_t` domain. The result can be seen in Table 6.5. Note that the "straight line" approach has been taken, i.e. the queries that are using the result of previous queries always takes the result needed. In principal, an automated tool would have to query every result it got back.

RACER query	**RACER reply**
`(individuals-related? nedit_t__i process__i hasTransitionClass)`	`T`
`(concept-instances (some transitionTo nedit_t))`	`(PROGRAMMER_TWITHNEDIT_EXEC_T__I STAFF_TWITHNEDIT_EXEC_T__I SECADM_TWITHNEDIT_EXEC_T__I SYSADM_TWITHNEDIT_EXEC_T__I)`
`(individual-fillers tom assign)`	`(SUPERVISOR_R__I)`
`(concept-parents supervisor_r)`	`((TESTER_R) (PROGRAMMER_R))`
`(individual-fillers programmer_r__i candoType)`	`(NEDIT_T__I PROGRAMMER_T__I)`
`(individual-fillers programmer_t__i candoType)`	`(PROGRAMMER_TWITHNEDIT_EXEC_T__I PROGRAMMER_TWITHMORE_EXEC_T__I PROGRAMMER_TWITHMOSML_T__I PROGRAMMER_TWITHMOSML_EXEC_T__I PROGRAMMER_TWITHNEDIT_T__I PROGRAMMER_TWITHMORE_T__I)`
`(individual-instance? PROGRAMMER_TWITHNEDIT_EXEC_T__I (some hasPermission executefile))`	`T`

Table 6.5: Question 5 answered for case study user `Tom` and type `nedit_t`

It can be seen that Tom via his role as supervisor, which dominates the programmer role, has permission to execute the executable of `nedit` which transitions into the `nedit` domain.

6.4.1 Miscellaneous queries

Various queries are put to the reasoner to show how the system responds.

Who has any access to documentation files?
`(concept-instances (some hasbasetype documentation_t))`

```
(MORE_TWITHDOCUMENTATION_T__I)
```

Is Tom a supervisor?

```
(individual-instance? Tom (some assign Supervisor_r))
T
```

Is Alice a supervisor?

```
(individual-instance? Alice (some assign Supervisor_r))
NIL
```

Who are programmers

```
(concept-instances (some assign Programmer_r))
(BOB TOM)
```

Who are authorized for the `nedit_t` type?

```
(concept-instances (some candotype nedit_t))
(SUPERVISOR_R__I PROGRAMMER_R__I)
```

Which domains interact with the `code_t` type?

```
(concept-instances (some hasBasetype code_t))
(NEDIT_TWITHCODE_T__I MOSML_TWITHCODE_T__I MORE_TWITHCODE_T__I)
```

It might surprise that the `more` program can interact with the `code_t` type, so it is interesting to see exactly which permissions the `more_t` domain have when interacting with the `code_t` type?

```
(individual-fillers MOSML_TWITHCODE_T__I haspermission)
(SEARCHDIR__I)
```

6.5 Summary

The chapter motivated the need for a verification of the knowledge base to check if security policies declarations were modeling appropriately in DL. The verification was done by informally rationale which concluded that the model presumably are able to model any SELinux policy under the conditions set by the thesis, since the verification did not find any faults after checking the different types of declarations. A catalog of queries were created to show the application of the DL model. Lastly, it was presented that the declarations of the security policy translates into many more axioms in the DL model, making it impossible to use the reasoner for queries regarding the full SELinux policy. A smaller subset of declarations was subtracted from the case study, and was used to extract some information regarding the case study.

The following chapter summarizes the project and discusses the encountered problems and gives resumes of some related work.

Chapter 7

Discussion

This chapter summarizes the status of the project and discusses other models and tools that have been found relevant to this thesis. Specifically it details the problems that have been experienced due to the size of the resulting knowledge base and tries to reason about the validity of the translation rules, modeled knowledge base and the queries regarding it.

7.1 Status

The case study has been transformed into a DL knowledge base by using the translation rules created by this thesis. The resulting knowledge base was compared to the policy to see if it modeled the case study. Furthermore, a catalog of queries was constructed for illustrating applications of the DL model and to verify the model.

As presented in Chapter 1 on page 1, creating a program domain in SELinux usually goes through a trial and error process wherein the system administrator configures the permissions for the program. There exist a Perl based tool, `Audit2allow` that collects all denied audit entries and creates appropriate type access declarations. Such a tool is useful to run after installing and running a new program, to help create the permissions needed by the program. The problem with `Audit2allow` is that it may create declarations that grant access to resources that the program should not be able to access. Using the model of the SELinux security policy the system administrator is able to execute queries that will help him comprehend the effects of the newly added permissions.

7.2 Problems

It became apparent that the model of the system was going to be quite large at a point when creating the translation rules. Looking at the translation rule for type allowance, for instance, it is clear that one declaration from SELinux can expand into an immense number of axioms in the knowledge base specification. Even though SELinux's domain definitions appear quite well-defined at a glance, this is mostly because of well named macros. Moreover, macros often use other macros, so a domain definition consisting of few declarations in the policy can very well expand up to many declarations "behind the scene", i.e. after SELinux's `checkpolicy` program has processed and created the combined policy file. This fact, along with the fact that one declaration expands to many axioms becomes an issue since the reasoner becomes sluggish even with a relative small knowledge base, the one used in Section 6.4 on page 68.

This brings forth the question of whether or not the model is sound. Since the knowledge base has been verified to model a (small) SELinux policy (Section 6.2 on page 62), with a complete set of declarations, the problem should not lie with the model, but the scalability of the system.

It should be noted that the reasoner used is that of version `1.7.23`, it was later observed that a newer version had been released, but it was not within the time-limit to retest on that release.

7.3 Related works

This section tries to give a small overview of papers and tools that have relevance to the topic of this thesis. The overview does not contain a high level of detail, but is meant as an appetizer for subjects that might be meaningful to lookup if interest has been aroused by this thesis.

7.3.1 Models

The article [Guttman et al., 2005] defines a model in linear temporal logic that sets out to model the SELinux policy as a labeled transition system in order to verify certain information flows in some states of the transition system by use of model checking.

Efforts are made in [Zanin and Mancini, 2004] where the authors seek to create a formalism to define an SELinux policy. They analyze the syntax of SELinux and presents a formal modal, *SELAC* (SELinux Access Control), that can be used for analysis. They create an accessibility algorithm based on their model that can determine whether a program has been given access to a specific permission on a specific object with a specific class, such a question is similar to Question 3 on page 66.

7.3.2 Tools

A tool named `Apol` shares some similarities with the tool developed through this thesis. It was developed by Tresys Technology, a company involved with the development of SELinux after it was released by NSA. The tool is also dependent on getting the policy file after the `checkpolicy` program has produced the complete policy in one file. The tool features an interface where the user can search through the policy looking up the different types, attributes etc. It also offers some analysis' upon the policy, such as domain transition analysis, file relabel analysis, type relationship summary and information flow analysis. See Figure 7.1 on page 77 for a screenshot displaying `Apol` in analysis mode. `Apol` distinguishes between direct information flow and indirect (transitive) information flow to feature two kinds of analysis. As the screenshot suggests, analysis upon the policy is time-consuming and they offer a way of time-limit the search for information flows.

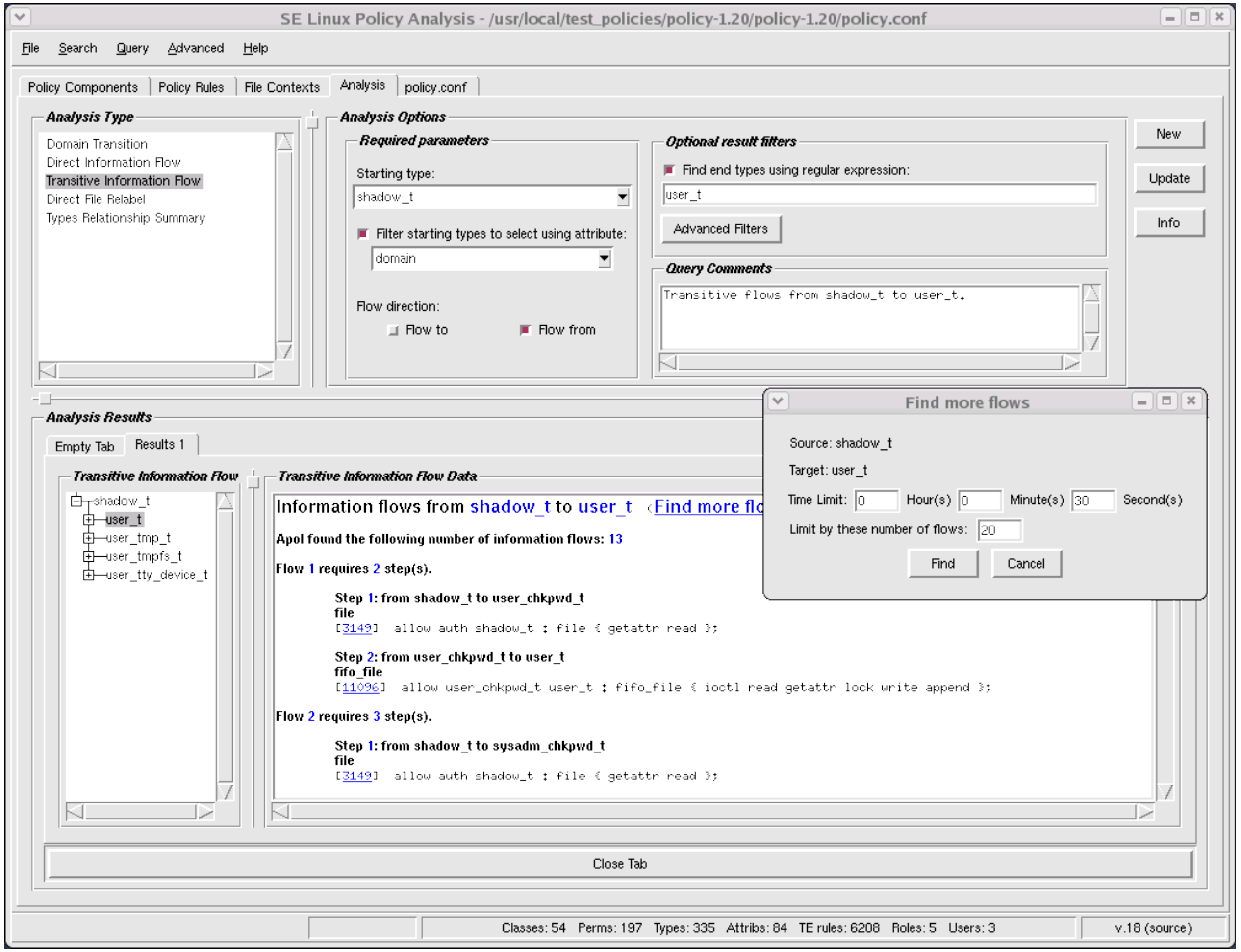
Tresys has developed several tools for SELinux and collectively released them as `SETools`. `Apol` though, is the tool whose feature-set resembles that of this thesis the most. Other tools include `SeDiff` - policy semantic diff tool for SELinux, `SeAudit` - searches, sorts and views audit messages from SELinux etc. [1].

[1] Can be found at `http://tresys.com/selinux/selinux_policy_tools.shtml`

7.4 Summary

The discussion chapter found that by using the translation rules as set forward by this thesis creates a knowledge base that is too large for the reasoner employed by this project. It also found that the problem does not lie with the knowledge base itself but with the size of the model, or perhaps an outdated reasoner, because the knowledge base has been validated to model the policy.

An overview of a number of related works were presented and as it will be seen in the next and final chapter, inspiration are found in these works, namely inspiration to expand the query catalog.



Starting type: shadow_t

Filter starting types to select using attribute: domain

Flow direction: Flow to [✓ Flow from]

Find end types using regular expression: user_t

```
Transitive flows from shadow_t to user_t.
```

```
shadow_t
  user_t
  user_tmp_t
  user_tmpfs_t
  user_tty_device_t
```

Information flows from shadow_t to user_t (Find more flo

Apol found the following number of information flows: 13

Flow 1 requires 2 step(s).

```
        Step 1: from shadow_t to user_chkpwd_t
        file
        [3149]  allow auth shadow_t : file { getattr read };

        Step 2: from user_chkpwd_t to user_t
        fifo_file
        [11096]  allow user_chkpwd_t user_t : fifo_file { ioctl read getattr lock write append };
```

Flow 2 requires 3 step(s).

```
        Step 1: from shadow_t to sysadm_chkpwd_t
        file
        [3149]  allow auth shadow_t : file { getattr read };
```

Source: shadow_t
Target: user_t
Time Limit: 0 Hour(s) 0 Minute(s) 30 Second(s)
Limit by these number of flows: 20

Classes: 54 Perms: 197 Types: 335 Attribs: 84 TE rules: 6208 Roles: 5 Users: 3 v.18 (source)

Figure 7.1: Screenshot from Apol

Chapter 8

Conclusion

The motivation for this project was strongly linked with that of *Security–Enhanced Linux* (SELinux), namely that the use of SELinux comes at a price in the complexity of configuration and maintenance of such a system. The hypotheses of this thesis was to test if a language such as *Description Logic* (DL) was able to model the security controls found in SELinux. The inspiration for the project ([Chen Zhao and Lin, 2005]) modeled *Role-Based Access Control* in DL but since RBAC plays a relative small part in the security controls of SELinux it was interesting to see if a complete model of SELinux could be created. An analysis of the concepts found in SELinux showed that *Type-Enforcement* (TE) played the main role of the actual security policy of SELinux and that each declaration found in the policy could be assigned a translation rule to create the necessary DL statements that would model each declaration.

Throughout the thesis a recurring example has been brought forward to show the different phases of the modeling. The case study exemplified a small hierarchy of user groups (*roles*) that together formed a software team (Chapter 1.3 on page 5). The case study was first implemented into a running SELinux distribution in order to determine exactly which declarations were needed in the policy to have the case study running on a SELinux system (Section 2.4 on page 24). The resulting policy was analyzed and a model that covered SELinux's policy was developed. Every SELinux declaration were given a translation rule into DL. The procedure for using the translation rules on the case study were given

(Section 4.3 on page 48) and queries regarding the case study were made to extract information that showcased the usefulness of the model (Section 6.4 on page 68).

It has been found that the objective of the thesis has been solved such that it has been shown that DL can model the security controls found in SELinux and useful queries can be made regarding it. The problem with the current status is that the size of the resulting knowledge base is on of such magnitude that the reasoning tool, *RACER*, cannot handle the model. It has been concluded that the model is reasonable and that the issue lies with the size or possibly the reasoner. There exists a risk that the model is flawed but the verification done in Section 6.2 on page 62 suggests otherwise. Nevertheless, there are points where further investigation would be useful.

The first item to investigate would be to run the knowledge base upon a new release of the reasoner to see if such efforts would be beneficial. Another strategy could be to further investigate alternative model formulations other than those explored in Section 4.4 on page 49 to see if any gains could be archived on that account.

It is clear that the query catalog could be expanded. It was not the main emphasis of this project to develop an all encompassing query catalog as that subject is quite large and complex as sketched up in the overview of related works Section 7.3 on page 74. Since the model of SELinux's security policy is complete with regard to ordinary programs (non-daemon programs) it is possible to develop information flow queries as discussed in the previous chapter, or other similar analysis' as defined in `Apol`.

One further formulation has been noticed that could be made to the model: Currently the permission concept is modeled as the concatenation of a permission and its security class. It is not possible to determine which class the permission concept has as base class without knowledge that the name has been created by concatenation. The formulation that would derive such information is similar to the hasBaseType role but is called hasBaseClass. It would function in the same way except that it would link a permission concept with its base security class.

Improvements upon the implementation might be available, e.g. optimizing the data structures used or optimizing the execution time, but in relation to the basis of the implementation, namely the SELinux's policy and the rules that translates them, such improvements are insignificant.

It would be good if a formal model check could be made in order to verify the validity of the translation rules. This project did not have time to go beyond informally inferring that the model was sound.

Lastly, as mentioned the query catalog can be expanded, but moreover functions that are able to create a set of queries that answers a general question, as seen in Section 6.3 on page 65, could be implemented. This would function similar as the macros seen in the SELinux policy, e.g. a function that answers Question 3 on page 66, could have the syntax `getDomainPermissions(domainName)` which would produce the needed RACER queries that could be entered into RICE.

The established rules for translating the SELinux policy models a knowledge base that has many possibilities for queries. The queries shown in this thesis is only the tip of the iceberg, and it is a separate discipline to create a set of queries that supports the many types of analysis as seen in Section 7.3 on page 74. The outlook of using the model of this thesis to support many forms of analysis is good, since the knowledge base models SELinux's declarations.

Appendix A

SELinux grammar

This appendix lists the grammar of SELinux's example policy. The grammar comes mostly from [Smalley, 2005], but not all production rules are stated there and some are outdated, so some declarations were found in the source of the SELinux parser, `checkpolicy` [NSA, 2005]. The grammar has been modified to only state what can be found in the example policy, since this thesis uses that policy as basis.

The grammar has been used in creation of the SML tool with few modifications.

The policy's top-level production is as follows

```
policy -> flask te_rbac users opt_contraints contexts
```

A.1 Flask

```
flask -> class_def | classes class_def
class_def -> CLASS identifier
initial_sids -> initial_sid_def | initial_sids initial_sid_def;
initial_sid_def -> SID identifier
```

```
access_vectors -> opt_common_perms av_perms;
opt_common_perms -> common_perms | empty
common_perms -> common_perms_def | common_perms common_perms_def;
common_perms_def -> COMMON identifier '{' identifier_list '}'
av_perms ->av_perms_def | av_perms av_perms_def
av_perms_def ->
    CLASS identifier '{' identifier_list '}'
  | CLASS identifier INHERITS identifier
  | CLASS identifier INHERITS identifier '{' identifier_list '}'
identifier_list -> identifier | identifier_list identifier
```

A.2 TE

```
te_rbac -> te_rbac_statement | te_rbac te_rbac_statement
te_rbac_statement -> te_statement | rbac_statement
te_statement -> attrib_decl |
                type_decl |
                type_transition_rule |
                type_change_rule |
                te_av_rule |
                te_assertion
rbac_statement -> role_decl |
                  role_dominance |
                  role_allow_rule


attrib_decl -> ATTRIBUTE identifier ';'


type_decl -> TYPE identifier opt_alias_def opt_attr_list ';'
opt_alias_def -> ALIAS aliases | empty
aliases -> identifier | '{' identifier_list '}'
opt_attr_list -> ',' attr_list | empty
attr_list -> identifier | attr_list ',' identifier
identifier_list -> identifier | identifier_list identifier
typealias_decl -> TYPEALIAS identifier ALIAS aliases ';'
typeattribute_decl -> TYPEATTRIBUTE identifier attr_list ';'

type_transition_rule -> TYPE_TRANSITION source_types target_types ':' classes new_typ
source_types -> set
target_types -> set
classes -> set
```

```
new_type -> identifier
type_change_rule -> TYPE_CHANGE set set ':' set identifier ';'

set ->    '*'
        | identifier
        | nested_id_set
        | '~' identifier
        | '~' nested_id_set
        | identifier '-' identifier
nested_id_set -> '{' nested_id_list '}'
nested_id_list -> nested_id_element | nested_id_list nested_id_element
nested_id_element -> identifier | '-' identifier | nested_id_set
```

```
te_av_rule -> av_kind source_types target_types ':' classes permissions ';'
av_kind -> ALLOW | AUDITALLOW | DONTAUDIT
source_types -> set
target_types -> set
classes -> set
permissions -> set
```

A.3 RBAC

```
role_decl -> ROLE identifier TYPES types ';'
types -> set

role_dominance -> DOMINANCE '{' roles '}'
roles -> role_def | roles role_def
role_def -> ROLE identifier ';' | ROLE identifier '{' roles '}'

role_allow_rule -> ALLOW current_roles new_roles ';'
current_roles -> set
new_roles -> set

role_transition_rule -> ROLE_TRANSITION current_roles types new_role ';'
current_roles -> set
types -> set
new_role -> identifier
```

A.4 Users

```
users -> user_decl | users user_decl
user_decl -> USER identifier ROLES set ';'
```

A.5 Constraints

```
opt_constraints -> constraints | empty
constraints -> constraint_def | constraints constraint_def
constraint_def -> CONSTRAIN classes permissions cexpr ';'
classes -> set
permissions -> set
cexpr -> '(' cexpr ')' | not cexpr | expr and expr | expr or expr |
          U1 op U2 | U1 op user_set | U2 op user_set |
          R1 role_op R2 | R1 op role_set | R2 op role_set
          T1 op T2 | T1 op type_set | T2 op type_set
not -> '!' | NOT
and -> '&&' | AND
or -> '||' | OR
op -> '==' | '!='
role_op -> op | DOM | DOMBY | INCOMP
user_set -> set
role_set -> set
type_set -> set
```

A.6 Security Contexts

```
contexts -> initial_sid_contexts fs_uses opt_genfs_contexts net_contexts
```

```
file_context_spec -> pathname_regexp opt_security_context |
                     pathname_regexp '-' file_type opt_security_context
file_type -> 'b' | 'c' | 'd' | 'p' | 'l' | 's' | '-'
opt_security_context -> <<none>> | user ':' role ':' type
user -> identifier
role -> identifier
type -> identifier
```

```
initial_sid_contexts -> initial_sid_context_def |
                        initial_sid_contexts initial_sid_context_def
initial_sid_context_def -> SID identifier security_context
security_context -> user ':' role ':' type
user -> identifier
role -> identifier
type -> identifier

fs_uses -> fs_use_def | fs_uses fs_use_def
fs_use_def -> FS_USE_XATTR fstype security_context ';' |
              FS_USE_TASK fstype security_context ';'
              FS_USE_TRANS fstype security_context ';'

opt_genfs_contexts -> genfs_contexts | empty
genfs_contexts -> genfs_context_def | genfs_contexts genfs_context_def
genfs_context_def -> GENFSCON fstype pathprefix '-' file_type security_context |
                     GENFSCON fstype pathprefix security_context
file_type -> 'b' | 'c' | 'd' | 'p' | 'l' | 's' | '-'

net_contexts -> opt_port_contexts opt_netif_contexts opt_node_contexts
opt_port_contexts -> port_contexts | empty
port_contexts -> port_context_def | port_contexts port_context_def
port_context_def -> PORTCON protocol port security_context |
                    PORTCON protocol portrange security_context
protocol -> 'tcp' | 'udp'
port -> integer
portrange -> port '-' port
opt_netif_contexts -> netif_contexts | empty
netif_contexts -> netif_context_def | netif_contexts netif_context_def
netif_context_def -> NETIFCON interface device_context packet_context
device_context -> security_context
packet_context -> security_context
opt_node_contexts -> node_contexts | empty
node_contexts -> node_context_def | node_contexts node_context_def

node_context_def -> NODECON ipv4_addr_def ipv4_addr_def security_context
            | NODECON ipv6_addr ipv6_addr security_context
ipv4_addr_def   -> number '.' number '.' number '.' number
ipv6_addr       -> IPV6_ADDR
```

A.7 Conditionals

```
bool_def -> BOOL identifier bool_val ';'
bool_val -> CTRUE | CFALSE
cond_stmt_def          -> IF cond_expr '{' cond_pol_list '}' cond_else
cond_else       : ELSE '{' cond_pol_list '}' | empty

cond_expr -> '(' cond_expr ')'
            | NOT cond_expr
            | cond_expr AND cond_expr
            | cond_expr OR cond_expr
            | cond_expr XOR cond_expr
            | cond_expr EQUALS cond_expr
            | cond_expr NOTEQUAL cond_expr
            | cond_expr_prim

cond_expr_prim -> identifier
cond_pol_list -> cond_pol_list cond_rule_def | empty

cond_rule_def -> cond_transition_def | cond_te_avtab_def
cond_transition_def -> TYPE_TRANSITION set set ':' set identifier ';'
                        | TYPE_CHANGE set set ':' set identifier ';'
cond_te_avtab_def   -> cond_allow_def
            | cond_auditallow_def
            | cond_auditdeny_def
            | cond_dontaudit_def
cond_allow_def  -> ALLOW set set ':' set set  ';'
cond_auditallow_def -> AUDITALLOW set set ':' set set ';'
cond_auditdeny_def -> AUDITDENY set set ':' set set ';'
cond_dontaudit_def -> DONTAUDIT set set ':' set set ';'
```

Appendix B

Source code

B.1 Abstract.sml

```
(****************************************************************)
(***************** SE Linux to Desciption Logic *****************)
(****************************************************************)
(*********************** Abstract syntax ************************)
(****************************************************************)
(******************** Alan Dickerson s991173 ********************)
(****************************************************************)

datatype Nested_id_list =
    Id_element of Nested_id_element
  | Id_listwithElement of Nested_id_list * Nested_id_element
and Nested_id_set =
   Id_list of Nested_id_list
and Nested_id_element =
    Id_nested of string
  | MinusId of string
  | Id_set_nested of Nested_id_set
;

datatype Set =
    Asterisk
  | Id of string
  | Id_set of Nested_id_set
  | TildeId of string
  | TildeSet of Nested_id_set
  | IdMinusId of string * string
;

datatype Role_def =
    Role of string
  | CompRole of string * Roles
and Roles =
    Def of Role_def
  | Comp of Roles * Role_def
;

datatype Cond_expr =
   Id_con of string
  | Cond_NOT of Cond_expr
  | Cond_AND of Cond_expr * Cond_expr
  | Cond_OR of Cond_expr * Cond_expr
  | Cond_XOR of Cond_expr * Cond_expr
  | Cond_EQ of Cond_expr * Cond_expr
  | Cond_NEQ of Cond_expr * Cond_expr
;

datatype Bool_val=
    TRUE
  | FALSE
;

datatype Decl =
      Common_perms_def of string * string list
    | Class_def of string
    | Class_def_perms of string * string list
    | Class_def_inherit  of string * string
    | Class_def_inherit_perms of string * string * string list
    | Attrib_decl of string
    | Type_decl of string * string list * string list
    | Typealias_decl of string * string list
    | Typeattribute_decl of string * string list
    | Type_transition_rule of Set * Set * Set * string
    | AllowDecl of Set * Set * Set * Set
    | AuditAllowDecl of Set * Set * Set * Set
    | Role_decl of string * Set
    | Role_dominance of Roles
    | Role_allow_rule of Set * Set
    | Role_transition of Set * Set * string
    | User_decl of string * Set
    | If of Cond_expr *  Block * Block
    | Bool_def of string * Bool_val
    | SKIP
withtype Block = Decl list;
;

type Policy = Decl list;
```

B.2 Lexer.lex

```
(*********************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(*********************************************************************)
(********************** Lexing specification **********************)
(*********************************************************************)
(********************* Alan Dickerson s991173 *********************)
(*********************************************************************)

{
 open Lexing Gram;

 (*(message, loc1, loc2)*)
 exception LexicalError of string * int * int

 fun lexerError lexbuf s = raise LexicalError
                (s, getLexemeStart lexbuf, getLexemeEnd lexbuf);

(* Scan keywords as identifiers and use this function to      *)
(* distinguish them.                                           *)
 fun keyword s =
     case s of
         "common"             => COMMON
       | "class"              => CLASS
       | "inherits"           => INHERITS
       | "attribute"          => ATTRIBUTE
       | "allow"              => ALLOW
       | "type"               => TYPE
       | "typeattribute"      => TYPEATTRIBUTE
       | "alias"              => ALIAS
       | "type_transition"    => TYPE_TRANSITION
       | "type_change"        => TYPE_CHANGE
       | "auditallow"         => AUDITALLOW
       | "dontaudit"          => DONTAUDIT
       | "neverallow"         => NEVERALLOW
       | "role"               => ROLE
       | "types"              => TYPES
       | "dominance"          => DOMINANCE
       | "role_transition"    => ROLE_TRANSITION
       | "user"               => USER
       | "roles"              => ROLES
       | "constrain"          => CONSTRAIN
       | "sid"                => SID
       | "fs_use_xattr"       => FS_USE_XATTR
       | "fs_use_task"        => FS_USE_TASK
       | "fs_use_trans"       => FS_USE_TRANS
       | "genfscon"           => GENFSCON
       | "portcon"            => PORTCON
       | "netifcon"           => NETIFCON
       | "nodecon"            => NODECON
       | "not"                => NOT
       | "and"                => AND
       | "or"                 => OR
       | "xor"                 => XOR
       | "u1"                 => U1
       | "u2"                 => U2
       | "r1"                 => R1
       | "r2"                 => R2
       | "t1"                 => T1
       | "t2"                 => T2
       | "dom"                => DOM
       | "domby"              => DOMBY
       | "incomp"             => INCOMP
```

```
        | "bool"                => BOOL
        | "if"                  => IF
        | "else"                => ELSE
        | "true"                => CTRUE
        | "false"               => CFALSE
        | "typealias"           => TYPEALIAS
        | _                     => IDENTIFIER s;
}

rule Token = parse
    [` ` `\t` `\n` `\r`]   { Token lexbuf } (* skip *)
  | [`0`-`9`]+             { case Int.fromString
                                  (getLexeme lexbuf) of
                        NONE   => lexerError lexbuf "internal␣error"
                      | SOME i => INT i
                  }
  | [`a`-`z``A`-`Z`][`a`-`z``A`-`Z``0`-`9``_`]*
                           { keyword (getLexeme lexbuf) }
  | "/"[`a`-`z``A`-`Z``0`-`9``_``.``-``/`]*     { PATH }
  | [`0`-`9``a`-`f`]*`:`[`0`-`9``a`-`f`]*`:`[`0`-`9``a`-`f``:``.`]* {
      IPV6_ADDR }
  | "#"                    { SkipToEndLine lexbuf; Token lexbuf }
  | `*`                    { ASTERISK }
  | `~`                    { TILDE }
  | `-`                    { MINUS }
  | `,`                    { COMMA }
  | `:`                    { COLON }
  | `;`                    { SEMI  }
  | `{`                    { LBRACK }
  | `}`                    { RBRACK }
  | `(`                    { LPAR }
  | `)`                    { RPAR }
  | "=="                   { EQUALS  }
  | "!="                   { NOTEQUAL }
  | "&&"                   { AND }
  | "||"                   { OR }
  | "^"                    { XOR }
  | `!`                    { NOT }
  | `.`                    { DOT }
  | eof                    { EOF }
  | _               { lexerError lexbuf "Illegal␣symbol␣in␣input" }
and SkipToEndLine = parse
    [`\n` `\r`]            { () }
   | (eof | `\^Z`)         { () }
   | _                     { SkipToEndLine lexbuf }
;
```

B.3 Gram.grm

```
%{
(*****************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(*****************************************************************)
(********************* Grammar specification **********************)
(*****************************************************************)
(********************* Alan Dickerson s991173 *********************)
(*****************************************************************)

open Abstract;

%}

%token <int> INT
%token <string> IDENTIFIER
%token ATTRIBUTE TYPE ALIAS TYPE_TRANSITION TYPE_CHANGE TYPEATTRIBUTE
%token ALLOW AUDITALLOW DONTAUDIT NEVERALLOW
%token ROLE TYPES DOMINANCE ROLE_TRANSITION COMMON CLASS INHERITS
%token USER ROLES
%token CONSTRAIN BLOCK IF ELSE CTRUE CFALSE NOT AND OR XOR PATH
%token SID
%token FS_USE_XATTR FS_USE_TASK FS_USE_TRANS
%token GENFSCON PORTCON NETIFCON NODECON IPV6_ADDR
%token ASTERISK TILDE MINUS COMMA COLON SEMI LBRACK RBRACK DOT
%token EOF

%token NOT AND OR EQUALS NOTEQUAL
%token U1 U2 R1 R2 T1 T2  TYPEALIAS
%token DOM DOMBY INCOMP BOOL
%token LPAR RPAR

%left OR /* lowest precedence */
%left XOR
%left AND
%right NOT
%left EQUALS NOTEQUAL /* highest precedence  */

%start Policy
%type <Abstract.Decl> Decl
%type <Abstract.Decl list> Block Decls opt_else
%type <Abstract.Policy> Policy
%type <Abstract.Set> Set
%type <Abstract.Nested_id_list> Nested_id_list
%type <Abstract.Nested_id_set> Nested_id_set
%type <Abstract.Nested_id_element> Nested_id_element
%type <Abstract.Role_def> Role_def
%type <Abstract.Roles> Roles
%type <Abstract.Cond_expr> Cond_expr
%type <Abstract.Bool_val> Bool_val

%type <string list> opt_alias_def aliases identifier_list opt_attr_list
      attr_list

/* Dummy types, they are skipped */
%type <Abstract.Decl> Cexpr Op Role_op Security_context Portrange Ipv4
%%

Policy:
    /* empty */                                  { [] }
  | Decl Policy  EOF                             { $1 :: $2 }
;

```

```

Decl :
    COMMON IDENTIFIER LBRACK identifier_list RBRACK    {
        Common_perms_def($2, $4)}
  | CLASS IDENTIFIER                                   { Class_def $2}
  | CLASS IDENTIFIER LBRACK identifier_list RBRACK     { Class_def_perms
      ($2,$4)}
  | CLASS IDENTIFIER INHERITS IDENTIFIER               {
      Class_def_inherit($2,$4)}
  | CLASS IDENTIFIER INHERITS IDENTIFIER LBRACK identifier_list RBRACK

  | ATTRIBUTE IDENTIFIER SEMI                          { Attrib_decl $2
      }
  | TYPEATTRIBUTE IDENTIFIER attr_list SEMI            {
      Typeattribute_decl ($2, $3) }
  | TYPE IDENTIFIER opt_alias_def opt_attr_list SEMI   { Type_decl($2,$3
      ,$4) }
  | TYPEALIAS IDENTIFIER ALIAS aliases SEMI            { Typealias_decl(
      $2, $4)}
  | TYPE_TRANSITION Set Set COLON Set IDENTIFIER SEMI  {
      Type_transition_rule($2,$3,$5,$6) }
  | ALLOW Set Set COLON Set Set SEMI                   { AllowDecl($2,$3
      ,$5,$6) }
  | AUDITALLOW Set Set COLON Set Set SEMI              { AuditAllowDecl(
      $2,$3,$5,$6) }
  | ROLE IDENTIFIER TYPES Set SEMI                     { Role_decl($2,
      $4) }
  | DOMINANCE LBRACK Roles RBRACK                      { Role_dominance
      $3}
  | ALLOW Set Set SEMI                                 { Role_allow_rule
      ($2, $3) }
  | USER IDENTIFIER ROLES Set SEMI                     { User_decl($2,
      $4) }
  | IF Cond_expr Block opt_else                        { If($2, $3, $4)
            }
  | BOOL IDENTIFIER Bool_val SEMI                      { Bool_def ($2,$3
      ) }

  | SEMI                                               { SKIP }
  | TYPE_CHANGE Set Set COLON Set IDENTIFIER SEMI      { SKIP }
  | DONTAUDIT Set Set COLON Set Set SEMI               { SKIP }
  | NEVERALLOW Set Set COLON Set Set SEMI              { SKIP }
  | ROLE_TRANSITION Set Set IDENTIFIER SEMI            { SKIP }
  | CONSTRAIN Set Set Cexpr SEMI                       { SKIP }
  | SID IDENTIFIER                                     { SKIP }
  | SID IDENTIFIER Security_context                    { SKIP }
  | FS_USE_XATTR IDENTIFIER Security_context SEMI      { SKIP }
  | FS_USE_TASK IDENTIFIER Security_context SEMI       { SKIP }
  | FS_USE_TRANS IDENTIFIER Security_context SEMI      { SKIP }
  | GENFSCON IDENTIFIER PATH MINUS IDENTIFIER Security_context

  | GENFSCON IDENTIFIER PATH Security_context
      { SKIP }
  | PORTCON IDENTIFIER INT Security_context            { SKIP }
  | PORTCON IDENTIFIER Portrange Security_context      { SKIP }
  | NETIFCON IDENTIFIER Security_context Security_context

  | NODECON Ipv4 Ipv4 Security_context                           { SKIP
    }
  | NODECON IPV6_ADDR IPV6_ADDR Security_context       { SKIP }
;


opt_alias_def:
    /* empty */                                         { [ ] }
  | ALIAS aliases                                       { $2 }
;

aliases:
    IDENTIFIER                                          { [ $1 ] }
  | LBRACK identifier_list RBRACK                       { $2 }
;

identifier_list:
    IDENTIFIER                                          { [ $1 ] }
  | identifier_list IDENTIFIER                          { $2::$1 }
;

opt_attr_list:
    /* empty */                                         { [ ] }
  | COMMA attr_list                                     { $2 }
;
attr_list:
    IDENTIFIER                                          { [ $1 ] }
  | attr_list COMMA IDENTIFIER                          { $3::$1 }
;

Set:
    ASTERISK                                            { Asterisk }
  | IDENTIFIER                                          { Id $1 }
  | Nested_id_set                                       { Id_set $1 }
  | TILDE IDENTIFIER                                    { TildeId $2 }
  | TILDE Nested_id_set                                 { TildeSet $2 }
  | IDENTIFIER MINUS IDENTIFIER                         { IdMinusId($1,
    $3)}
;

Nested_id_set:
   LBRACK Nested_id_list RBRACK                         { Id_list $2 }
;
Nested_id_list:
    Nested_id_element                                   { Id_element $1
      }
  | Nested_id_list Nested_id_element                    {
    Id_listwithElement($1,$2) }
;

Nested_id_element:
    IDENTIFIER                                          { Id_nested $1
        }
  | MINUS IDENTIFIER                                    { MinusId $2 }
  | Nested_id_set                                       { Id_set_nested
        $1 }
;

Roles:
    Role_def                                            { Def $1}
  | Roles Role_def                                      { Comp ($1, $2)}
;

Role_def:
    ROLE IDENTIFIER SEMI                                { Role $2 }
  | ROLE IDENTIFIER LBRACK Roles RBRACK                 { CompRole($2,$4
      ) }
;

Bool_val:
    CTRUE
                  { TRUE }
  | CFALSE
                { FALSE }
;

Block:
   LBRACK Decls RBRACK                { $2 }
;

opt_else:
    /* empty */                       { [ ] }
|   ELSE Block                        { $2 }
;

Decls:
    Decl                                        { [$1] }
  | Decl Decls                                  { $1::$2 }
;


Cond_expr:
    IDENTIFIER                       { Id_con $1 }
  | LPAR Cond_expr RPAR              { $2 }
  | NOT Cond_expr                    { Cond_NOT $2 }
  | Cond_expr AND Cond_expr          { Cond_AND( $1, $3) }
  | Cond_expr OR Cond_expr           { Cond_OR( $1, $3)}
  | Cond_expr XOR Cond_expr          { Cond_XOR( $1, $3) }
  | Cond_expr EQUALS Cond_expr       { Cond_EQ($1, $3)}
  | Cond_expr NOTEQUAL Cond_expr     { Cond_NEQ($1,$3) }
;

expr:
    LPAR Cexpr RPAR                                             { SKIP }
  | NOT Cexpr                                                     { SKIP
      }
  | Cexpr AND Cexpr                                               { SKIP
      }
  | Cexpr OR Cexpr                                              { SKIP }
  | U1 Op U2                                                    { SKIP }
  | U1 Op Set                                                   { SKIP }
  | U2 Op Set                                                   { SKIP }
  | R1 Role_op R2
      { SKIP }
  | R1 Op Set                                                   { SKIP }
  | R2 Op Set                                                   { SKIP }
  | T1 Op T2                                                  { SKIP }
  | T1 Op Set                                                   { SKIP
       }
  | T2 Op Set                                                 { SKIP }
;

Op            : EQUALS                            { SKIP }
              | NOTEQUAL                  { SKIP }
              ;
Role_op       : Op
                      { SKIP }
              | DOM                                       { SKIP }
              | DOMBY                             { SKIP }
              | INCOMP                            { SKIP }
              ;

Security_context:
    IDENTIFIER COLON IDENTIFIER COLON IDENTIFIER


;

Portrange:
INT MINUS INT
                             { SKIP }
;

Ipv4      :
INT DOT INT DOT INT DOT INT                              { SKIP }
;
```

B.4 makeparser.bat

```
rem Creation of Scanner/Parser
mosmlc -c Abstract.sml
mosmllex Lexer.lex
mosmlyac -v Gram.grm
mosmlc -c -liberal Gram.sig Gram.sml
mosmlc -c Lexer.sml
mosml parse.sml
```

B.5 parse.sml

```
(*****************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(*****************************************************************)
(********************* Parsing specification **********************)
(*****************************************************************)
(********************* Alan Dickerson s991173 *********************)
(*****************************************************************)

app load ["Location", "Nonstdio", "Gram", "Lexer"];
open Abstract;

(* Fancy parsing from a file; show the offending program piece*)
(* on error                                                   *)
fun parseExprReport file stream lexbuf =
    let val expr =
            Gram.Policy Lexer.Token lexbuf
            handle
               Parsing.ParseError f =>
                   let val pos1 = Lexing.getLexemeStart lexbuf
                       val pos2 = Lexing.getLexemeEnd lexbuf
                   in
                       Location.errMsg (file, stream, lexbuf)
                                      (Location.Loc(pos1, pos2))
                                       "Syntax error."
                   end
             | Lexer.LexicalError(msg, pos1, pos2) =>
                   if pos1 >= 0 andalso pos2 >= 0 then
                       Location.errMsg (file, stream, lexbuf)
                                      (Location.Loc(pos1, pos2))
                                       ("Lexical error: " ^ msg)
                   else
                       (Location.errPrompt ("Lexical error: " ^
                                                 msg ^ "\n\n");
                        raise Fail "Lexical error");
    in
        Parsing.clearParser();
        expr
    end
    handle exn => (Parsing.clearParser(); raise exn);

(* Create lexer from instream *)
fun createLexerStream (is : BasicIO.instream) =
    Lexing.createLexer (fn buff => fn n
                            =>
                            Nonstdio.buff_input is buff 0 n)
(* Parse a program from a file, with error reporting *)
fun parsef file =
    let val is   = Nonstdio.open_in_bin file
        val expr= parseExprReport file is (createLexerStream is)
                  handle exn => (BasicIO.close_in is; raise exn)
    in
        BasicIO.close_in is;
        expr
    end;
```

B.6 auxiliary.sml

```
(*******************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(*******************************************************************)
(********************** Auxiliary functions **********************)
(*******************************************************************)
(********************* Alan Dickerson s991173 *********************)
(*******************************************************************)

(* Takes a list and empties it over into the set
 * addToSet: a' list * set -> (b list * set)
 *)
fun addToSet ([], set)    = ([], set)
  | addToSet (x::xs, set) =
        let val set= Binaryset.add(set, x)
        in addToSet(xs, set)
        end;

fun insertIntoSetMap(attMap, _, []) = attMap
  | insertIntoSetMap(attMap, data, k::keys) =
  let
    val _ = case peek attMap k of
            SOME result =>
                let val attSet = Binaryset.add(result, data)
                in insert attMap (k, attSet)
                end
          | NONE => let
                        val attSet = Binaryset.empty String.compare;
                        val attSet = Binaryset.add(attSet, data)
                    in
                        insert attMap (k, attSet)
                    end
    in insertIntoSetMap(attMap, data, keys)
    end

  ;

fun lookupAlias(key, aliasMap) =
        let
            val items = case peek aliasMap key of
                SOME result => Binaryset.listItems result
              (* It was not an alias, then it is what it is *)
              | NONE        => [key]
        in
            items
        end


(* Lookup the typename in the attribute table to determine if it is a
    set*)
fun lookupAttribute(key, attMap, aliasMap) =
        let
            val items = case peek attMap key of
                SOME result => Binaryset.listItems result
              (* It was not an atribute identifier, was it an alias?*)
              | NONE        => lookupAlias(key, aliasMap)
        in
            items
        end
;


(* Determine all permissions in the system *)
```

```
fun getAllPerms([]) = []
  | getAllPerms((_,set)::ms) = (Binaryset.listItems set)@getAllPerms(ms)
;

fun lookupClassPerms([],_,_) =  []
  | lookupClassPerms(c::cs,map,permSet) =
    let
        val set = case peek map c of
            SOME result => result
          | NONE => Binaryset.empty String.compare;
    in
        Binaryset.intersection(permSet, set)::lookupClassPerms(cs,map,
            permSet)
    end
;

fun createCart (_, []) = []
  | createCart (str, x::xs) = concat(x::[str])::createCart(str, xs)
;

(* string list * string set list -> *)
fun createClassPerms (_, []) = []
  | createClassPerms ([], _) = []
  | createClassPerms (c::cs, p::ps) =
let
val perms = Binaryset.listItems p
in
 createCart(c,perms)::createClassPerms(cs,ps)
end
```

B.7 testFunctions.sml

```
(******************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(******************************************************************)
(*********************** Testing functions ***********************)
(******************************************************************)
(********************* Alan Dickerson s991173 *********************)
(******************************************************************)

val Ts = ["a", "b" ,"c"];
val Tt = ["d", "e"];
val testBools = [("a",true), ("b",false) ,("c",true)];
val testSet =Binaryset.empty String.compare;
fun printNamedSet [] = []
  | printNamedSet ((name, theSet)::xs) = (name, Binaryset.listItems
      theSet)::printNamedSet xs;

fun printTable table = printNamedSet (listItems table);

fun printSetlist [] = []
  | printSetlist (r::rs) = Binaryset.listItems(r)::printSetlist(rs);
```

B.8 stringFunctions.sml

```
(**********************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(**********************************************************************)
(*********************** String functions ************************)
(**********************************************************************)
(********************* Alan Dickerson s991173 *********************)
(**********************************************************************)

val addSpace = fn x => concat[x," \n"];
val addComma = fn x => concat[x,", "];
val addILine = fn x => concat[x,"  I\n"];
val addI     = fn x => concat[x,"  I"];

(* TBOX functions *)

(* inclString: Concept * Concept * Role -> string
* the strings have the form as:
* (implies File_type Attribute)
* (implies CProgrammer (some candoType Java_exec_t))
*)
fun inclString(c1,c2, "")   = concat(["(implies ", c1, " ", c2, ")","\n"])
  | inclString(c1,c2, role) =
  (* self identifier is sometimes used to denote when a type has a
   * permission upon itself. It is always used in conjunction with a role
   * since the Self concept does not exist.
   *)
      concat(["(implies ", c1, " (some ", role, " ", c2, "))","\n"])

;
(* incl: conceptList * conceptList * Role -> string list
*)
(* Provides a one to many mapping between first and second list*)
fun incl(_,[], _) = []
  | incl([],_, _) = []
  | incl(c1::c1s, c2::c2s, "") = inclString(c1,c2,"")::incl([c1],c2s,"")
  | incl(c1::c1s, c2::c2s, role) = inclString(c1,c2,role)::incl([c1],c2s,role)
;

exception ListsLengthError;

(* Provides a one to one mapping between first and second list*)
fun incl2(_,[], _) = []
  | incl2([],_, _) = []
  | incl2(c1::c1s, c2::c2s, role) =
    if length (c1::c1s) <> length (c2::c2s) then raise ListsLengthError else
        inclString(c1,c2,role)::incl2(c1s,c2s,role)

;



fun withString(_,[]) = []
  | withString([],_) = []
  | withString(t1::t1s, t2::t2s) =
  let
    val t2mod = if t2 = "self" then t1 else t2
  in
```

```
    concat([t1,"With",t2mod])::withString([t1],t2s)@withString(t1s,t2::
        t2s)
  end
;

fun targetsDecl (_, [])= []
  | targetsDecl ([], _)= []
  | targetsDecl (t1::ts, targetTypes)=
  let val withStrings = withString([t1], targetTypes)
  in
      incl([t1], withStrings, "targetsType")@targetsDecl(ts,targetTypes)
  end
;


fun replaceSelf (_, []) = []
  | replaceSelf (t1, t2::ts) =
  let
    val typeMod = if t2 = "self" then t1 else t2
  in
    typeMod::replaceSelf(t1,ts)
  end
;

fun hasBaseTypeDecl (_, [])= []
  | hasBaseTypeDecl ([], _)= []
  | hasBaseTypeDecl (t1::ts, t2::t2s)=
  let
    val targetTypes = replaceSelf(t1, t2::t2s)
    val withStrings = withString([t1],targetTypes)
    in
      incl2(withStrings, targetTypes, "hasBaseType")@hasBaseTypeDecl(ts,
        t2::t2s)
  end
;
(*
mosml {more,self}

mosmlWithmore
mosmlwithmosml
(implies mosml_exec_tWithmore_exec_t (some hasBaseType more_exec_t))
(implies mosml_exec_tWithmosml_exec_t (some hasBaseType self))
*)




fun hasPermissionDecl (_,[])= []
  | hasPermissionDecl ([],_)= []
  | hasPermissionDecl (withStrings, pc::pcs)=
  let fun createPerms([],perms) = []
      | createPerms (w1::ws,perms) = incl([w1], perms, "hasPermission")
          @createPerms(ws, perms)
    in
      createPerms(withStrings, pc)@hasPermissionDecl(withStrings,pcs)
  end
;


fun transitionToDecl ([],_)= []
  | transitionToDecl (w1::ws, name)=
       incl([w1], [name], "transitionTo")@transitionToDecl(ws,name)

;

```

```

fun candoRoleDecl ([],_)= []
  | candoRoleDecl (r1::rs, targetRoles)=
       incl([r1], targetRoles, "candoRole")@candoRoleDecl(rs,targetRoles)

;

fun roleDominanceDecl([]) = []
  | roleDominanceDecl(r1::rs) =
  let
    val (name, roles) = r1
  in
    incl([name], roles,"")@roleDominanceDecl(rs)
  end
;


fun conceptInclusion([],_) = []
  | conceptInclusion(p::ps, Concept) = incl([p], [Concept], "")@conceptInclusion(ps, Concept)
;




(********* ABOX functions *********)


(* (instance name Concept) *)
fun cAssertString(name, concept) = concat(["(instance␣",name,"␣",concept,")","\n"]);



(* (related c1 c2 role) *)
fun rAssertString(c1, c2, role) = concat(["(related␣",c1,"␣",c2,"␣",role,")","\n"]);

fun roleAssertDecl(name, [],_) = []
  | roleAssertDecl(name, r::rs,role) = rAssertString(name,r,role)::roleAssertDecl(name, rs, role)
;


(* Provides a one to one mapping between first and second list*)
fun roleAssertDecl2(_,[], _) = []
  | roleAssertDecl2([],_, _) = []
  | roleAssertDecl2(c1::c1s, c2::c2s, role) =
    if length (c1::c1s) <> length (c2::c2s) then raise ListsLengthError else
        rAssertString(c1,c2,role)::roleAssertDecl2(c1s,c2s,role)
;



fun conceptInstances([]) = []
  | conceptInstances(c::cs) = cAssertString(addI c, c)::conceptInstances(cs)


fun targetsDeclAbox (_, [])= []
  | targetsDeclAbox ([], _)= []
  | targetsDeclAbox (t1::ts, targetTypes)=
  let val withStrings = withString([t1], targetTypes)
  in
      roleAssertDecl(addI t1, List.map addI (withStrings), "targetsType"
          )@targetsDeclAbox(ts,targetTypes)
  end
;

fun hasBaseTypeDeclAbox (_, [])= []
  | hasBaseTypeDeclAbox ([], _)= []
  | hasBaseTypeDeclAbox (t1::ts, t2::t2s)=
  let
    val targetTypes = replaceSelf(t1, t2::t2s)
    val withStrings = withString([t1],targetTypes);
  in
      roleAssertDecl2(List.map addI (withStrings), List.map addI (
          targetTypes), "hasBaseType")@hasBaseTypeDeclAbox(ts,t2::t2s)
  end
;




fun hasPermissionDeclAbox (_,[])= []
  | hasPermissionDeclAbox ([],_)= []
  | hasPermissionDeclAbox (withStrings, pc::pcs)=
  let fun createPerms([],perms) = []
      | createPerms (w1::ws,perms) = roleAssertDecl(addI w1, List.map
          addI (perms), "hasPermission")@createPerms(ws, perms)
    in
      createPerms(withStrings, pc)@hasPermissionDeclAbox(withStrings,pcs
          )
  end
;

fun transitionToDeclAbox ([],_)= []
  | transitionToDeclAbox (w1::ws, name)=
       rAssertString(addI w1, addI name, "transitionTo")::
           transitionToDeclAbox(ws,name)

;

fun roleDominanceDeclAbox([]) = []
  | roleDominanceDeclAbox(r1::rs) =
  let
    val (name, roles) = r1
  in
    roleAssertDecl(addI name, List.map addI(roles), "candoType")
        @roleDominanceDeclAbox(rs)
  end
;

fun candoRoleDeclAbox ([],_)= []
  | candoRoleDeclAbox (r1::rs, targetRoles)=
       roleAssertDecl(addI r1, List.map addI(targetRoles), "candoRole")
           @candoRoleDeclAbox(rs,targetRoles)

;
(********* File functions ********)



fun makeFileOut(fileName) = TextIO.openOut(fileName)    ;

fun write(out, []) = ()
  | write(out, s::ss) =     let val _ =   TextIO.output(out,s);
  in
      write(out,ss)
  end
;


fun appendToFile(fileName, s) =
  let
    val out = TextIO.openAppend (fileName)
  in
    write(out,s);
    TextIO.closeOut(out)
  end;


(* fun writeToFile(out, s) =     TextIO.output(out,s); *)
fun writeToSig(s) =  appendToFile("signature.krss",s);
fun writeToTbox(s) =  appendToFile("tbox.krss",s);
fun writeToAbox(s) =  appendToFile("abox.krss",s);



fun saveFile(out) =     TextIO.closeOut(out);

fun makeFile(filename,s) =
  let
    val out = TextIO.openOut(filename)
  in
    TextIO.output(out,s);
    TextIO.closeOut(out)
  end;


(********* Signature *********)
fun createAllPermsClass(ClassSet, PermMap) =
        let
            val PermList    = getAllPerms(listItems PermMap)
            val AllPermSet  = Binaryset.empty String.compare
            val AllPermSet  = Binaryset.addList(AllPermSet, PermList)

            val Classes     = Binaryset.listItems ClassSet;
            val lookupList  = lookupClassPerms(Classes, PermMap,
                    AllPermSet)
            val PermClass   = createClassPerms(Classes,lookupList)
        in
            List.concat PermClass
        end
;


fun writeWithString _  [] = []
  | writeWithString f (w::ws) =
    let
        val _ = writeToSig([f w])
    in
        writeWithString f ws
    end
;


fun createSignature(name, ClassSet, TypeSet, RoleSet, PermMap, AttSet,
    UserSet, WithSet) =
    let
        val kb = ["(in-knowledge-base ",name," ",name,"Abox",")\n\n"]
        val _ = writeToSig(kb)

        val classes = Binaryset.listItems(ClassSet)
        val types = Binaryset.listItems(TypeSet)

        val withStrings = Binaryset.listItems(WithSet)

        val roles = Binaryset.listItems(RoleSet)
        val users = Binaryset.listItems(UserSet)
        val attributes = Binaryset.listItems(AttSet)
        val permClass = createAllPermsClass(ClassSet, PermMap)
        val sigStart = ["(signature"]
        val _ = writeToSig(sigStart)
        (* Concepts *)
        val _ = writeToSig(["\t:atomic-concepts (Class Type Permission 
            CRole User Attribute\n"])
        val _ = writeToSig([";; Classes\n"]@List.map addSpace (classes))
        val _ = writeToSig([";; Types\n"]@List.map addSpace (types))

        val _ = writeToSig([";; \t\tWithTypes\n"])
        val _ = writeToSig(List.map addSpace (withStrings))

        val _ = writeToSig([";; Roles\n"]@List.map addSpace (roles))
        val _ = writeToSig([";; Attributes\n"]@List.map addSpace (
            attributes))
        val _ = writeToSig([";; Combined PermissionClasses\n"]@List.map
            addSpace (permClass))
        val _ = writeToSig([")"])

        (* Roles *)
        val _ = writeToSig(["\t\t:roles (\n\t\t\t(candoType   :range 
            Type)\n\t\t\t(targetsType :parent candoType)\n\t\t\t(
            hasPermission :range Permission)\n\t\t\t(hasAttribute :range
             Attribute)\n\t\t\t(hasTransitionClass :range Class)\n\t\t\t
            (hasBaseType :range Type)\n\t\t\t(candoRole :range CRole)\n\
            t\t\t(transitionTo :range Type)\n\t\t\t(assign :range CRole)
            \n\t\t\t)"])

        (* Individuals *)
        val _ = writeToSig(["\n\t\t:individuals (\n"])
        val _ = writeToSig([";; Classes\n"]@List.map addILine (classes))
        val _ = writeToSig([";; Types\n"]@List.map addILine (types))


        val _ = writeToSig([";; \t\tWithTypes\n"])
        val _ = writeToSig(List.map addILine (withStrings))

        val _ = writeToSig([";; Roles\n"]@List.map addILine (roles))
        (* Note that users are an instance and only a space is added *)
        val _ = writeToSig([";; Users\n"]@List.map addSpace (users))
        val _ = writeToSig([";; Attributes\n"]@List.map addILine (
            attributes))
        val _ = writeToSig([";; Combined PermissionClasses\n"]@List.map
            addILine (permClass)@[")"])

        val sigEnd = [")"]

    in
        writeToSig(sigEnd)
    end
;
```

B.9 main.sml

```
(*****************************************************************)
(****************** SE Linux to Desciption Logic ******************)
(*****************************************************************)
(************************* Interpreter *************************)
(*****************************************************************)
(********************* Alan Dickerson s991173 *********************)
(*****************************************************************)
load "Binaryset";
load "Polyhash";
load "Date"; load "Time";

val startTime = Date.toString (Date.fromTimeLocal (Time.now ())) ;

open Polyhash;

exception ElementNotFoundInList;
exception UnkownError;


use "parse.sml";

use "auxiliary.sml";
use "testFunctions.sml";

use "stringFunctions.sml";

(* Clear running outputfiles *)
makeFile("signature.krss",concat [";;␣Output␣started␣on␣",startTime,"\n"
    ]);
makeFile("tbox.krss","");
makeFile("abox.krss","");





(********* Data modeling *********)
type StringSet = string Binaryset.set;

type RoleMap = (string, string list) hash_table;
type ClassSet = StringSet;
type AttributeSet = StringSet;
type TypeSet = StringSet;
type RoleSet = StringSet;
type UserSet= StringSet;

type CommonMap = (string, StringSet) hash_table;
type BoolMap = (string, bool) hash_table;
type AttributeMap = (string, StringSet) hash_table;
type AliasMap = (string, StringSet) hash_table;
type PermissionMap = (string, StringSet) hash_table;

val parsetree = parsef "mini.conf";


val INITIALHASHSIZE = 5000;

val ClassSet          = Binaryset.empty String.compare;
val AttributeSet      = Binaryset.empty String.compare;
val TypeSet           = Binaryset.empty String.compare;
val WithSet           = Binaryset.empty String.compare;
val RoleSet           = Binaryset.empty String.compare;
val UserSet            = Binaryset.empty String.compare;
val Abox               = Binaryset.empty String.compare;
val Bools =
        mkPolyTable(INITIALHASHSIZE, ElementNotFoundInList) : BoolMap;
val AttMap =
        mkPolyTable(INITIALHASHSIZE, ElementNotFoundInList) : AttributeMap;

val AliasMap =
        mkPolyTable(INITIALHASHSIZE, ElementNotFoundInList) : AliasMap;

val CommonMap =
        mkPolyTable(INITIALHASHSIZE, ElementNotFoundInList) : CommonMap;
val PermissionMap =
        mkPolyTable(INITIALHASHSIZE, ElementNotFoundInList) : PermissionMap;



fun PT1h (Common_perms_def (name,list),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val commonSet = Binaryset.empty String.compare;
                val (_,commonSet) = addToSet(list,commonSet)
                val _ = insert commonList (name, commonSet)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end

|  PT1h  (Class_def name,classSet, permissionList, bools,commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let val classSet= Binaryset.add(classSet, name)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet,  aliasMap, attMap, roleSet, userSet)
            end
 | PT1h (Class_def_perms (name,list),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val classSet= Binaryset.add(classSet, name)
                val permSet = Binaryset.empty String.compare
                val (_,permSet) = addToSet(list,permSet)
                val _ = insert permissionList (name, permSet)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h (Class_def_inherit (name,inherit),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val classSet= Binaryset.add(classSet, name)
                val foundSet = valOf(peek commonList inherit)
                val _ = insert permissionList (name, foundSet)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h (Class_def_inherit_perms (name, inherit,list) ,classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
```

```
                val classSet= Binaryset.add(classSet, name)
                val permSet = Binaryset.empty String.compare;
                (* Find the inheritances *)
                val foundSet = valOf(peek commonList inherit)
                (* Make the other permissionset *)
                val (_,permSet) = addToSet(list,permSet)
                (* Make the union *)
                val permSet = Binaryset.union(foundSet,permSet)
                val _ = insert permissionList (name, permSet)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end

 | PT1h (Attrib_decl name, classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let val attributeSet= Binaryset.add(attributeSet, name)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h  (Type_decl(name, aliases, attributeNames),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val typeSet= Binaryset.add(typeSet, name)
                val aliasMap =
                if List.length aliases <> 0
                then insertIntoSetMap(aliasMap, name, aliases)
                else aliasMap
                val attMap = insertIntoSetMap(attMap, name, attributeNames)
            in (classSet, permissionList, bools, commonList,attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h  (Typeattribute_decl (name, attributeNames),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val attMap = insertIntoSetMap(attMap, name, attributeNames)
            in (classSet, permissionList, bools, commonList,attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h  (Typealias_decl (name, aliases),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val aliasMap = insertIntoSetMap(aliasMap, name, aliases)
            in (classSet, permissionList, bools, commonList,attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
| PT1h  (Role_decl (name, _),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val roleSet= Binaryset.add(roleSet, name)
            in (classSet, permissionList, bools, commonList,attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
| PT1h  (User_decl(name, _),classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val userSet= Binaryset.add(userSet, name)
            in (classSet, permissionList, bools, commonList,attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end




 | PT1h ( Bool_def(name, b) ,classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            let
                val theBool = if b = TRUE then true else false
                val _ = insert bools (name, theBool)
            in (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            end
 | PT1h (_ ,classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
            (classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
            ;

(* PT1: parsetree * Classes * Permissions * Bools * Common * AttributeMap -> Classes * Permissions * Bools * Common * AttributeMap *)
fun PT1 ([],classSet, permissionList, bools, commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) = (classSet, permissionList, bools,commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
  | PT1 (decl::parsetree,classSet, permissionList, bools,commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet) =
    let
        val (ClassSet, Perms, Bools, Common, AttSet, TypeSet, AliasMap, AttMap, roleSet, userSet) = PT1h(decl, classSet, permissionList, bools,commonList, attributeSet, typeSet, aliasMap, attMap, roleSet, userSet)
    in  PT1(parsetree,ClassSet, Perms, Bools, Common, AttSet, TypeSet, AliasMap, AttMap, roleSet, userSet)
    end
;

fun evaluateIf(Id_con name,bools) =
let
                val result = case peek bools name of
                SOME theBool => theBool
              (* To accomedate a flawed policy, set all unread bools to false *)
              | NONE      =>
                        let
                                val _ = TextIO.print(name^" was undefined, setting to false\n")
                        in
                                false
                        end
                        in
                                result
                        end
  | evaluateIf(Cond_NOT expr, bools) =
    let val bool = evaluateIf(expr, bools)
    in not bool
    end
  | evaluateIf(Cond_AND (e1,e2), bools) = evaluateIf(e1, bools) andalso evaluateIf(e2, bools)
  | evaluateIf(Cond_OR (e1,e2), bools) = evaluateIf(e1, bools) orelse evaluateIf(e2, bools)
  | evaluateIf(Cond_XOR (e1,e2), bools) =
    let
```

```
        val b1 = evaluateIf(e1, bools)
        val b2 = evaluateIf(e2, bools)
    in
        (b1 orelse b2) andalso not (b1 andalso b2)
    end
  | evaluateIf(Cond_EQ (e1,e2), bools) = evaluateIf(e1, bools) =
      evaluateIf(e2, bools)
  | evaluateIf(Cond_NEQ (e1,e2), bools) = (evaluateIf(e1, bools) <>
      evaluateIf(e2, bools))
;

fun handleNestedSet(Id_list name,universeSet, map, aMap, workSet) =
        let
            fun handleIDElement(Id_nested name,universeSet,map,aMap,
                workSet) =
                (* Add the name(s) to the working set *)
                let
                    val names =  lookupAttribute(name, map , aMap)
                    val workingSet = Binaryset.addList(workSet,names)
                in
                    workingSet
                end
            | handleIDElement(MinusId name,universeSet,map, aMap,workSet
                ) =
                    let

                        val excludeList =   lookupAttribute(name, map,
                            aMap)
                        val excludeSet  = Binaryset.empty String.compare
                        val excludeSet = Binaryset.addList(excludeSet,
                            excludeList)
                        val workSet = if Binaryset.isEmpty workSet
                                      then universeSet
                                      else workSet
                        val workSet = Binaryset.difference(workSet,
                            excludeSet)
                    in
                        workSet
                    end
            | handleIDElement(Id_set_nested name,universeSet, map, aMap,
                workSet) =  handleNestedSet(name,universeSet, map,aMap,
                workSet)

            fun handleIDList(Id_element name,universeSet, map, aMap,
                workSet) = handleIDElement(name,universeSet, map, aMap,
                workSet)
          | handleIDList(Id_listwithElement (list, element),universeSet,
               map, aMap,workSet) =
                let
                    val set1 = handleIDList(list,universeSet, map, aMap,
                         workSet)
                    val set2 = handleIDElement(element,universeSet, map,
                         aMap, set1)
                in
                    set2
                end
        in
                handleIDList(name,universeSet, map, aMap,workSet)
        end
;
```

```
fun evaluateTypeSet(Asterisk, TypeSet, AliasMap, AttMap) = Binaryset.listItems TypeSet
  | evaluateTypeSet(Id name, TypeSet, AliasMap, AttMap) = lookupAttribute(name, AttMap, AliasMap)

  | evaluateTypeSet(Id_set name, TypeSet, AliasMap, AttMap) =
        let
            val emptySet = Binaryset.empty String.compare
            val set =     handleNestedSet(name,TypeSet, AttMap,AliasMap, emptySet)
        in
            Binaryset.listItems set
        end
  | evaluateTypeSet(TildeId name, TypeSet, AliasMap, AttMap) =
                        let
                            val excludeList = lookupAttribute(name, AttMap, AliasMap)
                            val excludeSet = Binaryset.empty String.compare
                            val excludeSet = Binaryset.addList(excludeSet, excludeList)
                        in
                            Binaryset.listItems(Binaryset.difference(TypeSet, excludeSet))
                        end
  | evaluateTypeSet(TildeSet name, TypeSet, AliasMap, AttMap) =
                        let
                            val emptySet = Binaryset.empty String.compare
                            val excludeSet = handleNestedSet(name, TypeSet, AttMap, AliasMap, emptySet)
                        in

                            Binaryset.listItems (Binaryset.difference(TypeSet, excludeSet))
                        end
  | evaluateTypeSet(IdMinusId (n1, n2), TypeSet, AliasMap, AttMap) =
                         let
                            val keepList    = lookupAttribute(n1, AttMap, AliasMap)
                            val excludeList = lookupAttribute(n2, AttMap, AliasMap)
                            val keepSet     = Binaryset.empty String.compare
                            val excludeSet  = Binaryset.empty String.compare
                            val keepSet     = Binaryset.addList(keepSet, keepList)
                            val excludeSet  = Binaryset.addList(excludeSet,excludeList)
                        in
                            Binaryset.listItems (Binaryset.difference(keepSet, excludeSet))
                        end
;

(* A function to handle allow and auditallow
 * These two declarations have the same impact in the DL system
 *)
fun handleAllow(allowType,Ts, Tt, C, P, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
    let
        val SourceTypes = evaluateTypeSet(Ts, TypeSet, AliasMap, AttMap)
        val TargetTypes = evaluateTypeSet(Tt, TypeSet, AliasMap, AttMap)

        val Classes = evaluateTypeSet(C, ClassSet, AliasMap, AttMap)
        (* Determine all permissions in the system *)
        val PermSet = Binaryset.empty String.compare
        val PermList = getAllPerms(listItems PermMap)
        val PermSet     = Binaryset.addList(PermSet, PermList)
        (* Get the items referenced in the set P*)
        val Perms = evaluateTypeSet(P, PermSet, AliasMap, AttMap)
        val usedPermSet = Binaryset.empty String.compare
        val usedPermSet     = Binaryset.addList(usedPermSet, Perms)
        (* Lookup classes perms from Classes then take intersection for
            each element with Perms *)
        val lookupList = lookupClassPerms(Classes, PermMap, usedPermSet)
        (* Classes is the same length as the lookup list*)
        val Perms = createClassPerms(Classes,lookupList);

        val withStrings = withString(SourceTypes,TargetTypes);




        (* Debug info *)
        val debug = [";;␣",allowType, "␣"]@
        ["{"]@List.map addComma (SourceTypes)@["}␣"]@
        ["{"]@List.map addComma (TargetTypes)@["}:"]@
        ["{"]@List.map addComma (Classes)@["}␣"]@
        ["{"]@List.map addComma (evaluateTypeSet(P, PermSet, AliasMap,
            AttMap))@["}\n"]
        val _ = writeToTbox(debug)
        val _ = writeToAbox(debug)


        (* The permissions and withstrings will be added at a later time
         * to increase performance
         *)

         (* Update the used types with the created "With" types*)
        val WithSet = Binaryset.addList(WithSet, withStrings)
        val Tbox =
            conceptInclusion(withStrings, "Type")@

            targetsDecl(SourceTypes,TargetTypes)@
            hasBaseTypeDecl(SourceTypes, TargetTypes) @
            hasPermissionDecl(withStrings, Perms);
        val _= writeToTbox(Tbox)

        val Abox =
            conceptInstances(withStrings)@
            targetsDeclAbox(SourceTypes,TargetTypes)@
            hasBaseTypeDeclAbox(SourceTypes, TargetTypes)@
            hasPermissionDeclAbox(withStrings, Perms);
        val _= writeToAbox(Abox)
    in
        (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools,
            WithSet)
    end
;


(* evaluateRoles: Roles -> (string,string list) hash_table*)
fun evaluateRoles(roles, roleMap) =
let
    fun evaluateRoleDef(Role name, _) = [name]
      | evaluateRoleDef(CompRole (name,roles), roleMap) =
      let
        val roleList = evaluateRoles(roles, roleMap)
```

```
          val _ = insert roleMap (name,roleList )
        in
        roleList
        end
    fun determineRoleType(Def(name), roleMap) = evaluateRoleDef(name, roleMap)
      | determineRoleType(Comp(roles, def), roleMap) = evaluateRoles(roles, roleMap)@evaluateRoleDef(def, roleMap)
in
    determineRoleType(roles, roleMap)
end
;


(* PT2: parsetree * Attributes * Types * Bools-> Attributes * Types * Bools *)
fun PT2 ([], ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) = ( ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
  | PT2 (decl::parsetree, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
    let
        (* Since IF statements consists of blocks of declarations, the
        * helping function the helping function PT2h is defined here
        *)
        fun PT2h  (If (expr, b1, b2), ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                        let
                            val bool = evaluateIf(expr,Bools)
                            val (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                                if bool
                                then PT2(b1, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
                                else PT2(b2, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
                        in
                            (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
                        end
              | PT2h (Attrib_decl name, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                  let
                        (* Debug info *)
                        val _ = writeToTbox([";;␣attribute␣",name,"\n"])
                        val _ = writeToAbox([";;␣attribute␣",name,"\n"])

                        val Tbox = incl([name], ["Attribute"], "")
                        val _= writeToTbox(Tbox)
                        val Abox = [cAssertString(addI name, name)]
                        val _ = writeToAbox(Abox)
                  in
                        (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
                  end

              | PT2h (Type_decl (name, aliases , attr), ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                  let
                        (* Debug info *)
                        val aliasStringList = if List.length aliases<> 0 then ["alias␣"]@List.map addComma aliases else []
```

```
                    val _ = writeToTbox([";;␣type␣",name, ","]
                        @aliasStringList@List.map addComma attr@["\n"])
                    val _ = writeToAbox([";;␣type␣",name, ","]
                        @aliasStringList@List.map addComma attr@["\n"])

                    val Tbox =
                        incl([name], ["Type"], "")@
                        incl([name], attr, "hasAttribute")

                    val _= writeToTbox(Tbox)

                    val Abox =
                        [cAssertString(addI name, name)]@
                        roleAssertDecl(addI name,List.map addI attr, "
                            hasAttribute")
                    val _ = writeToAbox(Abox)

                in
                    (ClassSet, TypeSet, AliasMap, RoleSet, PermMap,
                        AttMap, Bools, WithSet)
                end
            | PT2h (Typeattribute_decl (name, attr),  ClassSet, TypeSet,
                  AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                let
                    (* Debug info *)
                    val _ = writeToTbox([";;␣typeattribute␣",name,","]
                        @List.map addComma attr@["\n"])
                    val _ = writeToAbox([";;␣typeattribute␣",name,","]
                        @List.map addComma attr@["\n"])

                    val Tbox =
                    incl([name], attr, "hasAttribute")
                    val _= writeToTbox(Tbox)

                    val Abox = roleAssertDecl(addI name,List.map addI
                        attr, "hasAttribute")
                    val _= writeToAbox(Abox)
                in
                    (ClassSet, TypeSet, AliasMap, RoleSet, PermMap,
                        AttMap, Bools, WithSet)
                end
            | PT2h (AllowDecl(Ts, Tt, C, P), ClassSet, TypeSet, AliasMap
                 , RoleSet, PermMap, AttMap, Bools, WithSet) =
                let
                    val result = handleAllow("allow",Ts, Tt, C, P,
                        ClassSet, TypeSet, AliasMap, RoleSet, PermMap,
                        AttMap, Bools, WithSet)
                in
                    result
                end
            | PT2h (AuditAllowDecl(Ts, Tt, C, P), ClassSet, TypeSet,
                AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                let
                    val result = handleAllow("auditallow",Ts, Tt, C, P,
                        ClassSet, TypeSet, AliasMap, RoleSet, PermMap,
                        AttMap, Bools, WithSet)
                in
                    result
                end

            | PT2h (Type_transition_rule(Ts, Tt, C, name), ClassSet,
                TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools,
                WithSet) =
                let
```

```
                    val SourceTypes = evaluateTypeSet(Ts, TypeSet,
                        AliasMap, AttMap)
                    val TargetTypes = evaluateTypeSet(Tt, TypeSet,
                        AliasMap, AttMap)
                    val Classes = evaluateTypeSet(C, ClassSet, AliasMap,
                         AttMap)
                    val withString = withString(SourceTypes,TargetTypes)
                    (* Update the used types with the created "With"
                        types*)
                    val WithSet = Binaryset.addList(WithSet, withString)

                    (* Debug info *)
                    val debug = [";; type_transition "]@
                    ["{"]@List.map addComma (SourceTypes)@["} "]@
                    ["{"]@List.map addComma (TargetTypes)@["}:"]@
                    ["{"]@List.map addComma (Classes)@["} "]@
                    [" ", name," \n"]
                    val _ = writeToTbox(debug)
                    val _ = writeToAbox(debug)

                    (* Withstrings will be be added later to improve
                        performance *)
                    val Tbox =
                    conceptInclusion(withString, "Type")@
                    hasBaseTypeDecl(SourceTypes, TargetTypes) @
                    incl([name], Classes, "hasTransitionClass")@
                    transitionToDecl(withString,name)

                    val Abox =
                    conceptInstances(withString)@
                    hasBaseTypeDeclAbox(SourceTypes, TargetTypes) @
                    roleAssertDecl(addI name,List.map addI Classes, "
                        hasTransitionClass")@
                    transitionToDeclAbox(withString,name)


                    val _= writeToTbox(Tbox)
                    val _= writeToAbox(Abox)

                in
                    (ClassSet, TypeSet, AliasMap, RoleSet, PermMap,
                        AttMap, Bools, WithSet)
                end
            | PT2h (Role_decl (name, AllowedTypeSet), ClassSet, TypeSet,
                AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
                let
                    val AllowedTypes = evaluateTypeSet(AllowedTypeSet,
                        TypeSet, AliasMap, AttMap)


                    (* Debug info *)
                    val debug = [";; role ", name, " "]@
                        ["{"]@List.map addComma (AllowedTypes)@["}\n"]
                    val _ = writeToTbox(debug)
                    val _ = writeToAbox(debug)

                    val Tbox =
                        incl([name], ["CRole"], "")@
                        incl([name], AllowedTypes, "candoType")

                    val Abox =
                        [cAssertString(addI name, name)]@
                        roleAssertDecl(addI name,List.map addI
                            AllowedTypes, "candoType")



            val _= writeToTbox(Tbox)
            val _= writeToAbox(Abox)

        in
        (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap,
            Bools, WithSet)
        end

    | PT2h (Role_dominance roles, ClassSet, TypeSet, AliasMap,
        RoleSet, PermMap, AttMap, Bools, WithSet) =
    let
        val RoleMap =
            mkPolyTable(50, ElementNotFoundInList) : (string,
                string list) hash_table;
        val _ = evaluateRoles(roles,RoleMap)
        val roleList = listItems RoleMap
        val (superName, _) = hd(roleList)
        (* Add the super role to the set of roles*)
        val RoleSet= Binaryset.add(RoleSet, superName)
        val Tbox  =
            incl([superName], ["CRole"], "")@
            roleDominanceDecl(roleList)
        val Abox  =  [cAssertString(addI superName, superName)]

        val _= writeToTbox(Tbox)
        val _= writeToAbox(Abox)
    in
     (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap,
         Bools, WithSet)
    end

    | PT2h (Role_allow_rule (Rs, Rt), ClassSet, TypeSet,
        AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
        let
            val SourceRoles = evaluateTypeSet(Rs, RoleSet,
                AliasMap, AttMap)
            val TargetRoles = evaluateTypeSet(Rt, RoleSet,
                AliasMap, AttMap)
            (* Debug info *)
            val debug =
                [";; (role)allow "]@
                ["{"]@List.map addComma (SourceRoles)@["} "]@
                ["{"]@List.map addComma (TargetRoles)@["}\n"]
            val _ = writeToTbox(debug)
            val _ = writeToAbox(debug)

            val Tbox  = candoRoleDecl(SourceRoles, TargetRoles)
            val Abox  = candoRoleDeclAbox(SourceRoles,
                TargetRoles)

            val _= writeToTbox(Tbox)
            val _= writeToAbox(Abox)
        in
        (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap,
            Bools, WithSet)
        end

    | PT2h (User_decl(name, roles),  ClassSet, TypeSet, AliasMap
        , RoleSet, PermMap, AttMap, Bools, WithSet) =
    let
         val roleStrings = evaluateTypeSet(roles, RoleSet,
             AliasMap, AttMap)

```

```
                (* Debug info *)
                val _ = writeToAbox(["; ;␣user␣", name,"␣roles␣"]@ ["{"]@List.map addComma (roleStrings)@["}\n"])

                val Abox = roleAssertDecl(name,List.map addI roleStrings , "assign")@
                        [cAssertString(name, "User")]

                val _ = writeToAbox(Abox)

            in
             ( ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
            end



            | PT2h (_ ,  ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) = ( ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
        val ( ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) = PT2h(decl,  ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
    in  PT2(parsetree, ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet)
    end
;

(*Parse first time*)
val (ClassSet, Perms, Bools, Common, AttSet, TypeSet, AliasMap, AttMap, RoleSet,UserSet)  =
    PT1(parsetree, ClassSet, PermissionMap, Bools,CommonMap,AttributeSet ,TypeSet, AliasMap, AttMap, RoleSet, UserSet) ;



(*Parse second time*)
val (ClassSet, TypeSet, AliasMap, RoleSet, PermMap, AttMap, Bools, WithSet) =
    PT2(parsetree,ClassSet, TypeSet, AliasMap, RoleSet, Perms, AttMap, Bools, WithSet);

val _ = writeToTbox(["; ;␣Policy␣travered␣successfully ,␣write␣the␣Classes ␣and␣Permissions" ])
val _ = writeToAbox(["; ;␣Policy␣travered␣successfully ,␣write␣the␣Classes ␣and␣Permissions" ])

fun finalizeKB(ClassSet, PermMap) =
    let
        val Types = Binaryset.listItems(TypeSet)
        val Classes = Binaryset.listItems ClassSet
        val permList = createAllPermsClass(ClassSet, PermMap)

        val _ = writeToTbox(["; ;␣Classes:" ])
        val _ = writeToTbox(conceptInclusion(Classes, "Class"))
                                val _ = writeToAbox(["; ;␣Permissions:" ])
        val _ = writeToTbox(conceptInclusion(permList, "Permission"))

        val _ = writeToAbox(["; ;␣Classes:" ])
        val _ = writeToAbox(conceptInstances(Classes))
        val _ = writeToAbox(["; ;␣Permissions:" ])
        val _ = writeToAbox(conceptInstances(permList))

    in
```

```
          ()
     end
;

(* Add the permissions and classes to the T- and Abox *)
val _ = finalizeKB(ClassSet,PermMap);

(* Done creating the Abox and Tbox *)






(* Collect used concepts and Abox individuals and create RACER signature
      *)
(* Create the signature *)

val _ = createSignature("softtest", ClassSet, TypeSet, RoleSet, PermMap,
      AttSet, UserSet, WithSet);

(* Get the time *)
val now = Date.toString (Date.fromTimeLocal (Time.now ())) ;

writeToSig(["\n;; Output finished: ",now,"\n\n"]);
```

Appendix C

Case study implementation in SELinux

The case study additions to the regular SELinux example policy is found in this Appendix.

C.1 RBAC

```
dominance { role supervisor_r {role programmer_r;  } };
dominance { role supervisor_r {role tester_r; } };

dominance { role tester_r { role member_r; } };
dominance { role programmer_r { role member_r; } };

allow supervisor_r { tester_r programmer_r member_r };
allow programmer_r { member_r };
allow tester_r { member_r };
```

C.2 Users

```
user Tom roles { supervisor_r };
user Alice roles { tester_r };
user Bob roles {programmer_r };
user John roles { member_r };
```

C.3 nedit

C.3.1 nedit.te

```
##########################
#
# Rules for nedit for Software team casestudy
# Author: Alan Dickerson  <s991173@student.dtu.dk>
#

# Create the necessary types
type nedit_t, domain;
type nedit_exec_t, exec_type, file_type;
type code_t, file_type;

# Define the role
role programmer_r types { nedit_t };

# Make it a full user
full_user_role(programmer);

# Transition to the nedit domain when executing the program
domain_auto_trans(userdomain, nedit_exec_t, nedit_t);

# Allow that any user can login to the programmer role
role_tty_type_change(user, programmer);

# nedit can read and create files in the code_t dir
allow nedit_t code_t:file create_file_perms;
allow nedit_t code_t:dir create_dir_perms;

# nedit recursively looksup the code dir, so it also searches the /
    var dir
allow nedit_t var_t:dir search;

### The below declarations define rules that allow nedit to run

#nedit can read its settings files
allow nedit_t default_t:dir rw_dir_perms;
allow nedit_t default_t:file rw_file_perms;

allow nedit_t programmer_devpts_t:chr_file { getattr read write
    ioctl };

# nedit uses the shared libraries
uses_shlib(nedit_t);

allow nedit_t bin_t:dir { getattr read };
allow nedit_t default_t:dir { getattr read search };
allow nedit_t default_t:file { getattr read };
allow nedit_t etc_runtime_t:file { getattr read };
allow nedit_t lib_t:file { getattr read };
allow nedit_t locale_t:dir search;
allow nedit_t locale_t:file { getattr read };
allow nedit_t self:unix_stream_socket { connect create getattr read
     write };
allow nedit_t proc_t:dir search;
allow nedit_t proc_t:file { getattr read };
allow nedit_t sbin_t:dir { getattr read };
allow nedit_t tmp_t:dir search;
allow nedit_t usr_t:file { getattr read };
allow nedit_t xdm_tmp_t:dir search;
allow nedit_t xdm_tmp_t:sock_file write;
allow nedit_t xdm_xserver_t:unix_stream_socket connectto;
allow nedit_t default_t:lnk_file read;

allow nedit_t user_evolution_home_t:dir getattr;
allow nedit_t user_fonts_t:dir getattr;
allow nedit_t user_gnome_secret_t:dir getattr;
allow nedit_t user_gnome_settings_t:dir getattr;
allow nedit_t user_home_t:dir getattr;
allow nedit_t user_mozilla_home_t:dir getattr;
allow nedit_t user_mplayer_home_t:dir getattr;
allow nedit_t user_thunderbird_home_t:dir getattr;

##### End of nedit
```

C.3.1.1 Reduced version

The `full_user_role` macro is substituted by `type programmer_t, domain;`.
The `role programmer_r types { nedit_t };` is substituted with `role programmer_r types { nedit_t programmer_t };`
and the macro `role_tty_type_change` has been deleted.

C.3.2 nedit.fc

```
# nedit
/usr/local/bin/nedit  system_u:object_r:nedit_exec_t
# code dir is labeled
/var/code system_u:object_r:code_t
# code files will be labled with code_t (excluding dirs)
/var/code.* -- system_u:object_r:code_t
```

C.4 mosml

C.4.1 mosml.te

```
##########################
#
# Rules for mosml for Software team casestudy
# Author: Alan Dickerson <s991173@student.dtu.dk>
#

type mosml_t, domain;
type mosml_exec_t, exec_type, file_type;

role tester_r types { mosml_t };

full_user_role(tester);
domain_auto_trans(userdomain, mosml_exec_t, mosml_t);

role_tty_type_change(user, tester);

allow mosml_t code_t:file r_file_perms;
allow mosml_t code_t:dir r_dir_perms;

allow tester_t code_t:dir search;

### The below declarations define rules that allow mosml to run

# mosml uses the shared libraries
uses_shlib(mosml_t);

allow mosml_t var_t:dir { search };

allow mosml_t self:process execmem;
allow mosml_t user_devpts_t:chr_file { ioctl read write };
allow mosml_t default_t:dir search;

# allows the user to use the console
allow tester_t user_devpts_t:chr_file { ioctl read write };
allow mosml_t tester_devpts_t:chr_file { ioctl read write };

##### End of mosml
```

C.4.1.1 Reduced version

The `full_user_role` macro is substituted by `type tester_t, domain;`.
The `role tester_r types { mosml_t };` is substituted with `role tester_r types { mosml_t tester_t };`
and the macro `role_tty_type_change` has been deleted.

C.4.2 mosml.fc

```
# mosml
# mosml binary
/var/code/mosml -- system_u:object_r:mosml_exec_t
```

C.5 more

C.5.1 more.te

```
# # # # # # # # # # # # # # # # # # # # # # # # #
#
# Rules for more for Software team casestudy
# Author: Alan Dickerson <s991173@student.dtu.dk>
#

type more_t, domain;
type more_exec_t, exec_type, file_type;
type documentation_t, file_type;

role member_r types { documentation_t more_t };

full_user_role(member);

# Make executing more_exec_t enter the more_t domain
domain_auto_trans(userdomain, more_exec_t, more_t);

role_tty_type_change(user, member);

allow more_t documentation_t:file r_file_perms;
allow more_t documentation_t:dir r_dir_perms;

# more can lookup in the var dir
allow more_t code_t:dir search;

### The below declarations define rules that allow more to run

# more uses the shared libraries
uses_shlib(more_t);


# allows the user to use the console
allow more_t user_devpts_t:chr_file { read write getattr ioctl };
allow more_t member_devpts_t:chr_file { read write getattr ioctl };

allow more_t etc_t:file { getattr read };
allow more_t lib_t:file { getattr read };
allow more_t locale_t:dir search;
allow more_t locale_t:file { getattr read };
allow more_t var_t:dir search;

##### End of more
```

C.5.1.1 Reduced version

The `full_user_role` macro is substituted by `type member_t, domain;`.
The `role member_r types { more_t };` is substituted with `role member_r types { more_t`

member_t };
and the macro `role_tty_type_change` has been deleted.

C.5.2 more.fc

```
# more
/bin/more -- system_u:object_r:more_exec_t
/var/code/doc.*  system_u:object_r:documentation_t
```

Bibliography

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The description logic handbook: theory, implementation, and applications.* Cambridge University Press, New York, NY, USA.

[Boebert and Kain, 1985] Boebert, W. and Kain, R. (1985). A practical alternative to hierarchical integrity policies. *Proceedings of the Eighth National Computer Security Conference.*

[Chen Zhao and Lin, 2005] Chen Zhao, Nuermaimaiti Heilili, S. L. and Lin, Z. (2005). Representation and reasoning on rbac: A description logic approach. In *ICTAC*.

[Guttman et al., 2005] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134.

[Haarslev and Möller, 2001] Haarslev, V. and Möller, R. (2001). Racer system description. In Goré, R., Leitsch, A., and Nipkow, T., editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 701–705. Springer-Verlag.

[Johnson, 1979] Johnson, S. C. (1979). Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA.

[Kernighan and Ritchie, 1977] Kernighan, B. W. and Ritchie, D. M. (1977). The m4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA.

[Lesk, 1975] Lesk, M. E. (1975). Lex - A Lexical Analyzer Generator. CSTR 39, Bell Laboratories.

[Levesque, 1984] Levesque, H. J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212.

[Loscocco and Smalley, 2000] Loscocco, P. and Smalley, S. (Oct. 2000). Integrating flexible support for security policies into the linux operating system. Technical report, NSA and NAI Labs.

[McCarty, 2004] McCarty, B. (2004). *SELINUX NSA's Open Source Security Enhanced Linux*. O'Reilly.

[Möller et al., 2003] Möller, R., Cornet, R., and Haarslev, V. (2003). Graphical interfaces for racer: Querying daml+oil and rdf documents. http://www.dina.kvl.dk/~sestoft/mosml.html.

[Möller et al., 2004] Möller, R., Cornet, R., Haarslev, V., and Wessel, M. (2004). Racer user's guide and reference manual version 1.7.19. Technical report, Concordia University and Techn. Univ. Hamburg-Harburg and University of Hamburg.

[NSA, 2005] NSA (2005). checkpolicy-1.28 source code. http://www.nsa.gov/selinux/archives/checkpolicy-1.28.tgz.

[Romanenko et al., 2005] Romanenko, S., Russo, C., and Sestoft, P. (2005). Moscow ml. http://www.dina.kvl.dk/~sestoft/mosml.html.

[Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29(2):38–47.

[Smalley, 2005] Smalley, S. (2005). Configuring the selinux policy. Technical report, NSA.

[Zanin and Mancini, 2004] Zanin, G. and Mancini, L. V. (2004). Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145, New York, NY, USA. ACM Press.