Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

A Study on Access Control, Trust Management, Constraints and Types of an EHR System

A Thesis Presented by

Hiep Minh Nguyen Vuong

To

the Graduate School

In partial Fulfillment of the

requirements

for the Degree of

Master of Science

In

Computer Science

Stony Brook University

December 2010

Stony Brook University

The Graduate School

Hiep Minh Nguyen Vuong

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Professor Yanhong Liu, Thesis Advisor,
Computer Science Department

Professor Michael Kifer,
Computer Science Department

Professor Scott Stoller,
Computer Science Department

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

A Study on Access Control, Trust Management, Constraints and Types of a EHR System

by

Hiep Minh Nguyen Vuong

Master of Science

in

Computer Science

Stony Brook University

2010

Cassandra is a role-based trust management language that utilizes Datalog extended with constraints to materialize security policies. A policy in Cassandra is a set of rules that controls activations and deactivations of roles, and regulates users' access to different information. This thesis considers a formal policy written in Cassandra for the proposed electronic health record (EHR) system of U.K. The policy was developed for ease of understanding and analysis. However, due to lack of real data and implementation, people cannot test and put their trust on the line. Patients often reject the ideas of sharing their information via a system if they cannot control access to their own data. Doctors, in addition, will not use the system if it takes too long to respond, or has too many restrictions and ends up being hard to use. In this project, we focus on analyses of the policy rules for completeness, consistency, and types based on the structures and the interaction between rules. Our analysis program automatically determines a number of errors and outputs types of policy's elements.

General Terms: Languages, System

Additional Key Words and Phrases: Analysis, Cassandra, Datalog, health care system, rules, trust management, type

Table of Contents

List of Figures . . . vi
List of Table . . . vii
Acknowledgments . . . viii

1 Introduction . . . **1**
1.1 Trust Management . . . 1
1.2 Trust Management Policy Analysis . . . 1
1.3 Outline of this thesis . . . 2

2 Related Work . . . **4**
2.1 Access Control . . . 4
2.2 Access Control Models . . . 4
2.3 Trust Management Language . . . 5
2.4 Cassandra . . . 8
2.5 EHR Policy in Cassandra . . . 5

3 Policy Rule Definition Analysis . . . **10**
3.1 Determining Matching Definitions . . . 10
3.2 Analysis . . . 11
3.2.1 Preprocessing . . . 11
3.2.2 Analysis Algorithm . . . 11
3.3 Analysis Results . . . 12
3.3.1 Non-local calls . . . 13
3.3.2 Errors Found . . . 14

4 Policy Element Type Analysis . . . **17**
4.1 Determine types . . . 17
4.2 Types Constraints . . . 18
4.3 Collecting Constraints . . . 20
4.3.1 Preprocessing . . . 20
4.3.2 Equality Constraints . . . 21
4.3.3 Sub-type Constraints . . . 22
4.3.4 Base Constraints . . . 23
4.3.5 Selection Constraints . . . 24
4.3.6 Constructor Constraints . . . 24
4.3 Solve Constraints . . . 25
4.4.1 Inference Rules . . . 25
4.4.2 Datalog Rules . . . 25

5 Datalog with Equality Constraints **27**
5.1 Performance Problems 27
5.2 Union-Find 27
5.3 Use of Union-Find 30
5.4 Summary on Results Analysis 31

6 Implementation **33**

7 Conclusion **34**

Bibliography **35**

List of Figures

2.1 This picture represents Lampson's model on access control…………………4

2.2 Policy languages comparison. This picture gives us a brief ideas about the pros and cons of some popular policy languages such as Cassandra, RT, EPAL, PeerTrust……………………………………………………………………..6

3.3 This picture represents the relations between different sub classes in the system, with the emphasis on local calls or calls from other classes. It also helps us to pinpoint which class is the most used, most needed by others. Knowing this information will let us focus our attention on upgrading, fixing the most used classes to refine our system…………………………………………...................14

5.2 An illustration of how Union-Find algorithm works. It briefly highlight the main ideas of optimizing space through three simple steps: Make set, Union and Find……………………………………………………………………………28

List of Tables

2.2: The development of trust management languages from 1996 to 2006. This picture shows us some popular trust management language from 1996 to 2006 and theirs authors……………………………………………………………………7

5.1: This table shows how much time it takes to solve the constraints…………27

Acknowledgments

Completing a thesis and finishing a master degree are steps of a long journey that one cannot make it on his own. I want to use this part here to express my deepest appreciation to the people that have walked along my side, helped me to focus and keep it together until the end of my journey.

Firstly, I want to thanks Professor Annie Liu for her advice, patience, guidance and constant support during the making of this thesis. Her experience contributes and flourishes my growth as an individual and as a graduated student. I have thoughts of giving up along the way, and this thesis would not be able to be put together without her supervision.

In addition, I also want to thanks my host family, the Meadows, who greet me warmly and treat me nicely during all these years. Thanks for giving me a home away from home, and strength, as well as belief to march forward and reach the end of the line.

Last but not least, I want to thanks my parents, my sister that provides me endless support and encouragement.

Chapter 1

Introduction

1.1 Trust Management

Trust management is not a completely new research area, nor it is already thoroughly look at. To begin with, a trusted system is a system where several classes use the same sets of shared resources; the system itself frequently have an access control mechanism to regulate all users' activities. Trusted systems can be encountered more and more in our daily life, where computers and newer technologies are pushed to replace the existing paper works, to save time and provide better services, with some of the most notable examples are online banking services, health records management, employment recruiting and management …

As the system needs to deal with data of different degrees of confidentiality, depending on the sensitivity of each piece of information, delicate data structures and well-thought access control algorithms are needed. This leads us to the general problems people have to face while constructing a trusted system: How can we handle the data to satisfy different security criteria? How well defined is our set of policy? Does it contain any conflicts or unneeded rules?

Depending on your trusted system, your set of security features can largely vary. The more classes of users your system handles, the more complex your policy should be to prevent any information leakage or abusing the ability to access and distribute private information.

As personal data are valuable and should be handled with care, a trusted system's main focus revolves around how well the system can manage "trust," – how well the system distribute/control/validate the users' requests based on his/her roles. Trusted systems' weaknesses are proportional to how well defined their sets of policies are. If the control policies have flaws and conflicts, the system may counter cases where outsiders can gain access to classified, sensitive or personal information. Hence, a lot of efforts are put into researches to construct a better and safer language.

1.2 Trust Management Policy Analysis

Building a system is just like constructing a house from scratch, finding your way out of a maze, or solving a mysterious puzzle. Trust management systems are often hard to build. The delicacy of its credentials management algorithm further adds on to the complexity of the rules based system. The interaction between classes is varied upon the scoop they are in. Selecting Cassandra list of rules for the NHS as a case study, we want to analyze its structure, examine its access control policies, getting under its complicated outer shield and parts, and thus gain a clearer look at the system as a whole.

The process of building a system with a trust management language such as Cassandra from limited number of rules may sound possible at first, but the degree of difficulty of such process will increase as the number of rules grows larger and larger. The relationships between classes, methods, variables will become more complex with each additional rule we add into the system. With the intention of alleviating the process's complexity, we will go through several steps of preparation before starting to analyze the EHR rules and policies.

A system needs three components to be complete: the input it takes in, the processing of data, and the output it gives back. We will apply the same structure to our steps of preparation. In each of our analysis, we want to answer three specific questions: "What is our input?" "How will this preparation help us to reduce the complexity in visualizing the final system?" and "What is the output that we can get?"

There are many ways to analyze the EHR policies. In this project, we focus on performing a 2-tier analysis: a policy source analysis to help us visualize the overall structure of the policy, and a type analysis to help us with the details about the needed variables in the policy. With this, we can get a sense of the large picture without tangle ourselves up in the web of details.

1.3 Outline of This Thesis

The rest of the thesis is structured as follows:

Chapter 2 is about related works. This chapter is a brief summary on trust management language and some of it key issues. We will first go over some of the most notable trust management languages and access control mechanism that are available at the moment.

Chapter 3 is about policy rule definition analysis. As the first tier of the analysis, it will help us to visualize the interaction graph between the

main components of the policy. It will tackle on the problems of determining the existing undefined calls. In addition, it will also give us an idea on how local/global calls can affected the hierarchy structure of EHR role based policy.

Chapter 4 is about policy element type analysis. As the second tier of the analysis, it will help us to see the underlining details of each component. This analysis is a vital step that we must go through in order to understand which kind of data is available and how can we assemble such data for a call to be possible.

Chapter 5 In this part, we will visit some problems on the running time of the program and how to solve/optimize this matter. Furthermore, we conclude on how effective this approach is and what we can do in the future to further optimize our time on other similar problems.

The last part is the thesis' bibliography.

Chapter 2

Related Work

2.1 Access Control

According to The US Department of Health and Human Services, an electronic health record system has to satisfy the following two conditions [10]:

• The system must be available right away to be used in different hospitals, and it must ensure a high quality service.

• The system must be able to protect the patients' privacy.

To satisfy the second requirement, different access control mechanisms are created. Access control is a process where requests are examined and filtered out, based on a set of policies. These policies are requirements that will give us details on how provide access. It answers the question: "In what situation, who can access what?" Access control is there to ensure that every request that needs the accession to some internal data is under the system's control. The system will grant requests that satisfy the set of policies, else the requests will be denied. Access control mechanism can be seen as a reference monitor that can process the incoming requests.

A simple model of access control [12, 18]:

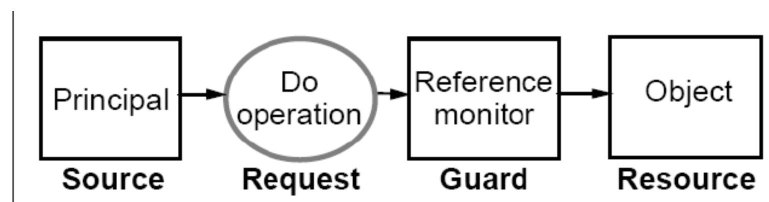

Figure 2.1: Lampson's access-control model [12]

Access control models are techniques for programmers to bring in everything and tie them together, hence create a trusted system. The three most popular Access control models are: Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC).

2.2 Access Control Models

Mandatory Access Control model was first introduced by David Bell and Leonard LaPadula, 1975 [17]. Mandatory access control can be summarized by the sentence of “no read up, no write down.” [19]. “No read up” tells us that a user can not have access to data of higher classification (represented through different security labels such as “normal”, “secret”…). “No write down” tell us that a user can read, but can not modify a lower classification. Even though the Mandatory Access Control model was widely used in the military field, it lacks the flexibility that a modern trusted system needs to have.

Discretionary Access Control was introduced by Butler Lampson in 1971 [20]. It presents the ideas of each object in the system has an owner. The owner can decide which user can access the files and grant him the request. Access control List / Access Control Matrix are used to record which users are granted the rights to access certain files. However, by doing this, the level of security takes a fall, and some new problems are created. A person now needs to keep an eye on the list of users who can access his file, who he previously granted the permission to do so. As the number of users scales up, the process of management becomes a real issue. Also, the users most of the time do not fully understand the importance of their data, and grant the wrong people the power to access sensitive and personal data that should be protected and keep intact. Redundancy is also a problem that we need to carefully look at.

Mandatory Access Control and Discretionary Access Control both have pros and cons, but we need more than that. Role Based Access Control was introduced in 1992 [21], with the idea of achieving the best of both worlds. This new model are said to be able to maintain a high level of security and flexibility, at the same time have the power to deal with problems of big scale. People start to get their hands on the ideas of roles and constrains. In this thesis, we will work with Cassandra, a Datalog based trust management language which are built on the idea of Role Based Access Control. In particular, we will examine an EHR system to conduct our study and analysis.

2.3 Trust Management Languages

With the development of various access control models comes the era of trust management languages. Many studies are conducted to craft a language that holds enough power to integrate into the real world. Up until this moment, researchers create and spend most of their time to study, analyze and polish the following two families of trust management language: the languages that are Datalog based, and those that are not.

	Cassandra	EPAL	KAoS	PeerTrust	Ponder	Protune	PSPL	Rei	RT	TPL	WSPL	XACML
Well-defined semantics	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No
Monotonicity	Yes	–	Yes	Yes	–	Yes	Yes	Yes	Yes	No	–	–
Underlying formalism	Constraint DATALOG	Predicate logic without quantifiers	Description logics	Constraint DATALOG	Object-oriented paradigm	Logic programming	Logic programming	Deontic logic, Logic programming, Description logics	Constraint DATALOG	–	–	–
Action execution	Yes (side-effect free)	Yes	No	Yes (only sending evidences)	Yes (access to system properties)	Yes	Yes (only sending evidences)	No	No	No	Yes	Yes
Delegation	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes (RT^D)	No	No	No
Type of evaluation	Distributed policies, Local evaluation	Local	Local	Distributed	Local	Distributed	Distributed	Distributed policies, Local evaluation	Local	Local	Distributed policies, Local evaluation	Distributed policies, Local evaluation
Evidences	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations	Credentials, Declarations	–	Credentials	Credentials	No	No
Negotiation	Yes	No	No	Yes	No	Yes	Yes	No	No	Yes	No (policy matching supported)	No
Result format	A/D and a set of constraints	A/D, scope error, policy error	A/D	A/D	A/D	Explanations	A/D	A/D[a]	A/D	A/D	A/D, not applicable, indeterminate	A/D, not applicable, indeterminate
Extensibility	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes

Figure 2.2: Policy languages comparison [5]

There are many languages that we could examine for example: Binder by John DeTreville [08], Lithium by Halpern and Weissman [13], RT by Li, Mitchell and Winsborough [14], and the OASIS framework [15] etc.

Name	Authors	Years
Policy Maker / Key Note	Blaze, Feigenbaum, et al.	1996 [16]
SPKI/SDSI	Frantz, Thomas, Rivest et al.	1997 - 1999 [04]
OASIS	Richard Hayton, Jean Bacon, and Ken Moody.	1998 [15]
RT	Ninghui Li, Mitchell and Winsborough.	2000 [14]
PSPL	Bonatti and Samarati.	2000 [09]
TPL	Herzberg, Mass, Michaeli et al.	2000 [07]
QCM / SD3	Trevor Jim, Carl Gunter.	2001 [32]
Ponder	Damianou.	2001 [22]
Binder	John DeTreville.	2002 [08]
PeerTrust	Nejdl, Olmedilla, et al.	2003 [31]
EPAL	Ashley, Hada et al.	2003 [24]
Rei	Kagal, Finin, and Joshi.	2003 [06]
KaOS	Uszok, Bradshaw, Jeffers et al.	2003 [25]
Lithium	Halpern and Weissman.	2003 [13]
xacml	Simon Godik.	2003 [26]
Cassandra	Moritz Y. Becker.	2004 [1]
WSPL	Anderson, A.H.	2004 [30]
RW	Zhang N.	2005 [23]
SecPAL	Becker, Gordon, and Fournet.	2006 [2]
DKAL	Yuri Gurevich, Itay Neeman.	2008 [27]
DKAL 2	Yuri Gurevich, Itay Neeman.	2009 [28]
TrustBuilder2	Adam J. Lee, Marianne Winslett, and Kenneth J. Perano.	2009 [29]

Table 2.2: A summary of development of trust management / policy languages and frameworks.

Among these languages, PolicyMaker [16] is the first to be introduced as a tool for assembling a trust management system. PolicyMaker's next version, KeyNote is a more secured and updated version of its predecessor where tighter verification algorithms are applied. QCM [32] is the first distributed "trust management system based on a declarative query language." As DKAL1, 2 [27, 28] are developed on the inspiration of SecPAL [2], they are said to have more expressive power than the previous.

2.4 Cassandra

In this project, we will look at Cassandra, a scalable language with enough flexibility, expressive power to cover a variety of securities policies and also smoothly manage a request's authorization process.

Cassandra is first introduced in 2005 by Moritz Y. Becker [01] at Cambridge University. The author proposed the security policy of EHR system based on the requirements of the NPfIT – the UK National Health Service's National Programme for Information Technology. Though the policy itself is sufficient enough to be used in a real world application, it still have flaws that can be discovered through further analysis.

After creating and improving Cassandra from 2004-2005, the author moves on and continues to explore this area. In 2006, SecPAL [02, 03] was born with the collaboration between Becker, Gordon, and Fournet. Recently, Becker focuses on improving SecPAL - Security Policy Assertion Language, through a series of research in 2009-1010. In compared to Cassandra, SecPAL possesses a more delicate credentials management system. In addition, SecPAL introduces proof trees for each answer it gives out, thus improving readability and tractability.

Since SecPAL is still in the developing phase with the latest extension introduced in September 2009, and Cassandra provides all the things we need to have, we decide to use Cassandra for a more accurate and stable analysis.

2.5 EHR Policy in Cassandra

The EHR policy expressed in Cassandra has four main components: the Spine, the Patient Demographic Service, the local hospital, and the local hospital's Registration Authority.

The Spine contains all the electronic health records. Both patients and clinicians can have access to the Spine. It has in total 137 rules. The Spine has the

following set of roles: *Clinician, Admin, Patient, Agent, and Third Party*. In order to access any information from the Spine, a user needs to activate his role first. Only one role can be activated at a time.

The Patient Demographic Service holds the data about the nationwide patient demographic information. It has the following set of roles: *Manager, Patient, Agent, and Professional User*. You have to follow the same rule of having one and only one role activated if you want to access the patient demographic information through the system.

Local hospital / heath organization is a local unit where it can provide its users seven roles such as: *Clinician, Caldicott-guardian, HR-manager, Receptionist, Patient, Agent, Ext-treating-clinician and Third-party*. Registration Authorities serve as the bridge between the Spine and the local health organizations. They can manage credentials and will provide them if asked by others. The only role available in this part is RA-manager. The RA-manager can register or deregister an individual with roles such as *manager*, *clinician*, *Caldicott-guardian …*

Six special predicates that contribute to the four general requests of performing an action, activating / deactivating a role, and requesting a credential:

permit (e,r)
canActivate(e, r)
hasActivated(e, r)
canDeactivate(e1,e2, r)
isDeactivated(e, r)
canReqCred(e1,e2.p(~e))

Chapter 3

Policy Rule Definition Analysis

In order to analyze the Electronic Health Record policy, we need to have brief ideas about how their components will interact with each other. Manually reading through about three hundred rules are tedious and time consuming. For such reasons, for the first part, we focus on analyzing the structure of the system by locating the definition of each predicate each time it is used by others. We want to know which classes use which classes. Do their interactions satisfy what we know about the Spine, the RAs etc…?

3.1 Determining Matching Definitions

To begin our analysis, we start by trying to identify what kind of data we have in hands, and how should we process them. If we know the structures of the inputs and the outputs, we can start thinking about how to extract out the needed information.

Input: In the proposed Electronic Health Record system, the inputs that we have are rules that are written in the language of Cassandra, in another words, Datalog with constraints.

Example:
'A4.1.3'
canDeactivate(cli1,cli2,Concealed-by-clinician(pat,id,start,end)) <-
hasActivated(cli1,Clinician(spcty1)),canActivate(cli1,ADB-treating-clinician(pat,group,spcty1)),canActivate(cli2,ADB-treating-clinician(pat,group,spcty2))

Output: We want the outputs to tell us the location information of calls in the input rules. We want to know if a call is valid or not: Is there a suitable definition available? If there is, where is that definition situated? Does it belong to the same classes, or does it need the information from somewhere else?

Example:
Rule A4.1.3: canDeactivate use:
'A4.1.3'
canDeactivate(cli1,cli2,Concealed-by-clinician(pat,id,start,end)) <-

hasActivated(cli1,Clinician(spcty1)),canActivate(cli1,ADB-treating-clinician(pat,group,spcty1)),canActivate(cli2,ADB-treating-clinician(pat,group,spcty2))

hasActivated
A1.1.4
canActivate
Not Defined
canActivate
Not local:
R2.1.1

When reading the outputs, we need to know the difference between:

• Not defined: there is no actual definition across all files.
• Not local: there is a definition elsewhere but not in input rule's component.

3.2 Analysis

3.2.1 Preprocessing

We will break each input rule into a collection of atom-size elements. We will then process each element to locate its actual location. This can be done by looping through different files to find matches. Depend on particular characteristics of each element; we can go to different files to search for matches. Some require a throughout search, some we only need to look into a specific file.

Furthermore, there are some key features of the system that we need to understand in order to perform our analysis:

• hasActivated(...) is not predefined. It will be created and inserted into the end of the file each time we make an activation. It will be treated more or less like a fact. In order to locate hasActivated(…), we look for its equivalent canActivate(…)

• Variables' name can change from rules to rules, in order to match them, we must decide if their types are the same or not.

• Functors not only must match the type of each other, they also must have the same number of arguments, as well as the same function name.

3.2.1 Analysis Algorithm

The first thing we need to do is to extract the conclusions' names and store them into a set. Conclusions' names are the methods' names available in our

system. If a predicate (represents a called method) whose name does not exist in the above set, and its name is not hasActivated, we can conclude that this method somehow is called by others while there is no specific definition for it. Hence, this is an undefined method. We will sort out these special cases to help identify the flaws in our rules system. Such flaws can be corrected by investigating on what are the actual usages and the scoop of the missing methods.

If a predicate whose name is in the set, we need to locate that specific predicate based on its name, number of arguments and the type of each argument. At this point, we will pay special attention to the scoop of the method. We will divide the conclusions' names into several sub-classes according to the program skeleton. We consider a match a "local" match if the predicate's name can be found in the same sub-class as the conclusion's name. Otherwise, it will be considered as a "Not local" match since they did not belong to the same scoop.

For a "Not local" call, we want to find all matches, so we will go through the entire set of rules instead of stopping after finding the first match. When this step is done, we can see if there's any ambiguity – where there are two or more "not local" methods, and there is no clear indications on which one we must pick.

We only pick the methods that satisfy all the requirements such as: same number of arguments, suitable types…

If there is no match at the end, we will also report it as Not Defined. (In this case, there exists a predicate with which it shares the same name, but they may have different numbers of arguments or their arguments' types may not match).

If there is a match or matches, we will clearly indicate if it is a local call or not.

In order to save time, we introduce a list whose elements are classes where the issuers come from. By doing this, if a call is a local call, we do not need to go through the whole set of rules to check for matches:

iss = {'"RA-ADB"':ra, '"PDS"':pds, '"Spine"':sp, '"ADB"':hos, 'ra':ra, 'org':hos, '"NHS"':rules}

3.3 Analysis Results

In EHR policy, we consider the following four components, according to the rules designed by Moritz Y. Becker. They are:

- A: rules for Addenbrooke’s Hospital
- R: rules for Addenbrooke’s Registration Authority
- S: rules for the Spine
- P: rules for the PDS - Patient Demographic Service

3.3.1 Non-Local Calls

Our analysis yields the following rules that contain calls to predicates that are defined by rules in other components:

A2.2.2 calls R2.1.1
S1.3.1 calls P2.1.1
A1.5.4 calls P2.1.1
S2.2.10 calls P2.1.1
P2.2.5 calls R2.3.1
A2.2.1 calls R2.1.1
S1.4.1 calls P2.1.1
S1.1.2 calls R2.1.1
S1.4.7 calls R2.3.1
S2.4.6 calls R3.1.1
S2.4.6 calls R3.1.2
R3.1.1 calls A3.2.2
R3.1.1 calls A3.2.1
R3.1.2 calls A3.5.1
R3.1.2 calls A3.5.2
A2.3.12 calls P2.1.1
P1.4.2 calls R2.1.1
A1.6.2 calls S1.4.1
S3.4.1 calls R3.1.2
S3.4.1 calls R3.1.1
S3.4.2 calls R3.1.2
S3.4.2 calls R3.1.1
S1.1.1 calls R2.1.1
P1.4.3 calls R2.2.1
S1.4.8 calls R2.3.1
P1.3.1 calls S1.4.1
P1.4.4 calls R2.2.1
A1.6.1 calls P2.1.1
A1.6.2 calls P2.1.1
P2.2.4 calls R2.3.1
P1.4.1 calls R2.1

We can observe that:

Out-degree for each node:
A calls methods from R,S,P and itself. Among the four sub-classes, its out-degree is the highest.
S calls methods from R, P and itself. P calls methods from S, R and itself.
R calls methods from A and itself, its out-degree is the lowest.

In-degree for each node:
R has the highest in-degree. (A,S,P,R).
S, P has the same in-degree. (A,P,S for S and A,S,P for P)
A has the lowest in-degree. (R,A)

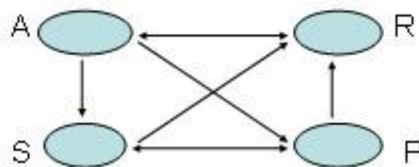

Figure 3.3: Policy Source Graph

We can see that the Spine S get called by A, P, S as the users of A, P, S can request information from the Spine. The local heath organization A calls methods from R,S,P and itself. The users at the local heath organization may want to get data from the Spine S, request credentials from Registration Authority R, or interact with the Patient Demographic Service P.

We also see that R has the highest in degree. This is true because the Spine, local health organizations, other RA and users can request the credentials from Registration Authority R. In addition, any users can access the information about Patient Demographic Service. (A,S,P for P).

3.3.2 Errors Found

There are calls that do not have matching definition. Our analysis yields the following where all of them are about deactivation except for the ones in bold.

isDeactivated(x,Register-patient(pat)) in S3.1.4
isDeactivated(y,Register-patient(pat)) in A3.7.4
isDeactivated(y,Register-patient(pat)) in S4.1.5
other-consent-to-group-treatment-requests(n,y,pat,org,group) in S2.4.12

isDeactivated(y,Register-patient(pat)) in A3.6.6
ra@"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in S1.5.2
"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in R1.2.4
isDeactivated(x,Register-HR-mgr(mgr)) in A1.3.6
isDeactivated(y,Register-patient(pat)) in S4.3.7
isDeactivated(x,Register-receptionist(rec)) in A1.4.6
isDeactivated(x,NHS-health-org-cert(org,start2,end2)) in R2.2.3
isDeactivated(y,Register-patient(pat)) in S1.4.13
isDeactivated(y,Register-patient(pat)) in S2.3.7
isDeactivated(x,Register-spine-admin(adm)) in S1.2.3
isDeactivated(y,Register-patient(pat)) in S4.2.6
isDeactivated(y,Register-clinician(cli,spcty)) in A3.5.6
isDeactivated(y,Register-patient(pat)) in S2.1.7
isDeactivated(x,Register-Caldicott-guardian(cg)) in A1.2.6
"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in P1.5.2
isDeactivated(x,Register-RA-manager(mgr)) in R1.1.6
ra@"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in A1.7.3
isDeactivated(y,Register-clinician(mem,spcty)) in A3.2.5
isDeactivated(z,Register-patient(pat)) in S2.2.8
isDeactivated(x,Register-PDS-manager(adm)) in P1.1.3
count-preofessional-user-activations(n,user) in P1.5.1
"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in A1.7.2
isDeactivated(x,Register-patient(pat)) in P1.3.4
isDeactivated(x,Register-patient(pat)) in P1.2.3
isDeactivated(x,NHS-health-org-cert(org,start2,end2)) in R2.1.3
isDeactivated(y,Register-patient(pat)) in A4.1.5
isDeactivated(x,Register-patient(pat)) in S1.3.3
isDeactivated(y,Register-patient(pat)) in S3.2.4
ra@ra.canActivate(cli2,Group-treating-clinician(pat,ra,org,group,spcty2)) in S4.2.10
"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in S1.5.1
isDeactivated(y,Register-patient(pat)) in A1.6.9
ra@ra.canActivate(cli1,Group-treating-clinician(pat,ra,org,group,spcty1)) in S4.2.10
isDeactivated(y,Register-patient(pat)) in A3.3.6
isDeactivated(y,Register-patient(pat)) in A4.2.6
isDeactivated(x,Register-patient(ag)) in P1.3.3
isDeactivated(x,Register-patient(pat)) in A1.5.6
isDeactivated(y,Register-patient(pat)) in S2.4.7
ra@"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in R1.2.5
isDeactivated(y,Register-patient(pat)) in A2.3.10

isDeactivated(x,Register-clinician(cli,spcty)) in A1.1.6
isDeactivated(y,Register-patient(pat)) in A2.3.20
ra@"NHS".hasActivated(x,NHS-registration-authority(ra,start,end)) in P1.5.3
isDeactivated(x,Spine-clinician(ra,org,spcty)) in S3.2.3
isDeactivated(y,Register-patient(pat)) in A2.1

Chapter 4

Policy Element Type Analysis

In order to fully understand a new class or a new data structure, we need to understand its structure and usage. In chapter 3, we get an overview on how each rule will be used and how they will interact with each other. In this analysis, we try to examine the structure of the components through its types. Such analysis will help us to understand better each component and give us ideas on what kind of data structures are needed.

In this part of the project, we want to analyze the types of all predicates and variables in the system. In order to complete this task, we will go through three main steps. We begin with constructing the needed constraints using rules as our inputs. Constraints are relations that link variables and their usages together. By solving these constraints in the next step, we will be able to identify most types in the system. Lastly, we will take care of special cases to come up with the final results.

4.1 Determining Types

The first step we need to consider is how to build type-related constraints from the input that we have. We will represent constraints as tuples. In a rule, atoms and functions are numbered from left to right, starting with one and increasing as much as needed.

Input: In the proposed EHR, the inputs that we have are rules that are written in the language of Cassandra, in another words, Datalog with constraints. Arguments can come from an atom or an atom's functor. Hence, we need to examine all of them.

Example:
'A4.1.3'
canDeactivate(cli1,cli2,Concealed-by-clinician(pat,id,start,end)) <-
hasActivated(cli1,Clinician(spcty1)),canActivate(cli1,ADB-treating-
clinician(pat,group,spcty1)),canActivate(cli2,ADB-treating-
clinician(pat,group,spcty2))

Output: The outputs will have the following form:
----varName: set of constraints and their types.

Example:
----y : set([('con', ('y', 'Clinician')), ('base', ('y', 'int')), ('base', ('y', 'date')), ('tuple', ('y', 'z1', 'z2', 'z3', 'z4', 'z5', 'z6', 'z7')), ('con', ('y', 'Patient'))])

4.2 Type Constraints

For each argument, we want to find out what its type is. The type of an element must be consistent in all cases.

Case 1: Maintaining consistencies between different rules.

'S1.5.3'
no-main-role-active(user) <-
count-agent-activations(n,user),**count-spine-clinician-activations**(n,user),count-spine-admin-activations(n,user),count-patient-activations(n,user),count-third-party-activations(n,user),n = 0

'S1.1.4'
count-spine-clinician-activations(count<u>,user) <-
hasActivated(u,Spine-clinician(ra,org,spcty)),u = user

Assume in S1.1.4, count<u>'s type is t1, user's type is t2.
Assume in S1.5.3, n's type is t3, user's type us t4.
In this case, we must have t1 equals t3, t2 equals t4.

Case 2: Maintaining consistencies within each rule.
'P1.4.3'
canActivate(**x**,Professional-user(**ra**,**org**)) <-
no-main-role-active(**cg**),**ra**.hasActivated(**x**,NHS-Caldicott-guardian-cert(**org**,**cg**,start,end)),canActivate(**ra**,Registration-authority()),Current-time() in [start,end]

Assume the type of the first x we encounter (canActivate(**x**, …)) is t1, then the type of the later x must also be t1.

The same rule applies for the rest such as **ra**, **org** and **cg**.

To do so, we assign a new type to an element only if it does not fit in with any existing type in our knowledge base. If we can find a suitable type in our knowledge base, we will assign the found type to the input element. The knowledge base must be updated as we move along. We need to keep track on the data structures of each element and update frequently as more information

becomes available through reasoning (a=b, a and b must have the same type) and cascading effects (a's type is t, e=(a,b,c,d), h=(a,f,g) → need to update the type of e and h). We must maintain consistencies as stated in the above cases while updating our knowledge base.

Main cases that must be considered:

1. x1 = y
   Assume type of x is tx1, type of y is t2, then tx1 must be the same as t2.
   In case type of x is unknown, set tx1 = t2.
   Example: n = 0
   → type of 0 is integer, hence type of n is also integer.

2. x2 = name(y1, y2, …., yn)
   y1 has type t1, y2 has type t2, … yn has type tn.
   In this case, tx2 = name(t1,t2,…tn)
   Example: what = (pat,ids,authors,groups,subjects,from-time,to-time)
   → what's type is a set of 7 elements. They can be of the same type or different types.

3. x3 = pi n_1(x2)
   x3 has type tx3. tx3 must be set to t1 since the type of x2 is
   tx2 = name(t1,t2,…tn)
   Example: pi7_1(what) = pat1
   → pi7_1(what) = pat. Assume pat's type is t1, pat1's type must also be set to t1.

Real rules will be more complicated in term of structures and variables' names. Hence, we need to establish a few more relations to construct a throughout analysis. We combine the following relations with the general cases above to scan for all existing types in our system.

Relations:
x = C(x1,x2,…,xn)
y = x

→ y = C(x1,x2,…,xn)
This relation is used to link case 1 and case 2 together.

x = C(x1,x2,…,xn)
y = pi n_1(x)

→ y = x1
This relation is used to link case 2 and case 3 together.

x = y
y = z

→ x = z
This relation is used to iterate through a list of case 1.

4.3 Collecting Type Constraints

4.3.1 Preprocessing

Take a look at the following rule:

```
'S1.5.3'
no-main-role-active(user) <-
    count-agent-activations(n,user),count-spine-clinician-activations(n,user),count-spine-admin-activations(n,user),count-patient-activations(n,user),count-third-party-activations(n,user),n = 0
```

Our indexing method suggests:
no-main-role-active: the 1st atom of S1.5.3
count-agent-activations: 2nd atom of S1.5.3
…
count-third-party-activations: 6th atom of S1.5.3

In the first round of iteration, as we read in each rule as input, we will return tuples of variables' type and their information. We will temporally assign a type t_n to each variable we encounter in the process, with n starts with 1 and keeps on increasing each time we assign a type. The variables' information such as where it comes from (the rule's index, the atom's index) and which argument it is (the argument's index) will be recorded in the output tuples.

The index of an argument in an atom is a tuple of up to two elements. If an argument's index is a pair, the first element represents the function's index, while the second element represents the argument's index in the list of arguments. If it is a singleton tuple, it simply represents the argument's index.

With the above example, we will return tuples such as:

(‘S1.5.3’, 1, (1), t100) : S1.5.3 is the name of the rule; 1 is the atom’s index which indicates this is the first atom in the rule: no-main-role-active; 1 is the argument’s index which indicates this argument is the first argument (user) we encounter in no-main-role-active; t100 is the type that we assign to argument user (assume this is the 100th time we assign a type to an argument)

Similarly, we will have:

(‘S1.5.3’, 2, (1), t101): argument n from count-agent-activations now has type t101
(‘S1.5.3’, 2, (2), t102): argument user from count-agent-activations now has type t102
(‘S1.5.3’, 3, (1), t103): argument n from count-spine-clinician-activations now has type t103
…
(‘S1.5.3’, 6, (2), t110): argument user from count-third-party-activations now has type t110

These tuples are used to uniquely identify each variable and its type. In the next few steps, we will generate a variety of constraints to help us gain more information about the system. In this case, there are five main types of constraints that we need to look at. They are:

- **eq(x,y):** equality constraints, x and y must be the same type

- **sub(x,y)**: sub-type constraints, x is a subtype of y or x equals to y.

- **base(x, t)**: basic facts, type of x is t (int, string…)

- **sel(x,y)**: x is of type ti if y is of type tuple(t1..ti..tk)

- **con(x,C,y1,...,yk)**: x is of type C(t1,...,tk) if y1..yk are of types t1..tk.

4.3.2 Equality Constraints

At this point, we can clearly see that one of the flaw of the present set of constraints is that we have the same argument holds different type values. Argument *user* from the above example has t100, t102,…, t110 as its types. Hence, in the second steps of iteration, we will output constraints that are made to deal with this specific issue. These constraints are tuples whose elements are the types that are equivalent to each other. This step helps us to narrow down and

combine some types together, hence lightening the load of the unknown type variables.

With the above example, for argument user, we will output the following tuple:
eq(t100, t10)
eq(t100, t104)
…
eq(t108, t110)
t100, t102,…, t110 are equivalent to each other.

Similarly, for argument n, we will output:
eq(t101,t103)
eq(t101,t105)
…

Furthermore, inspecting on rules' constraints lead us to more local equalities, such as in this example:
(R2.3.5)
canReqCred(e, "RA-ADB".hasActivated(x,
NHS-health-org-cert(org, start, end))) <-
hasActivated(y, NHS-clinician-cert(org, cli, spcty, start2, end2)),
Current-time() in [start2, end2],
e=cli

Assume type of e is t1, type of cli is t8, we will have the following tuples:
eq(t1,t8)

This will help us to further strengthen our eq(x,y) tuples.

4.3.3 Sub-Type Constraints

Since we also have to consider cases where other rules already use or define a particular atoms/functions, in the next step, we must take care of type consistencies between different rules. For example, we have rule such as S1.1.4 that defines the atom count-spine-clinician-activations:

```
'S1.1.4'
count-spine-clinician-activations(count<u>,user) <-
    hasActivated(u,Spine-clinician(ra,org,spcty)),u = user
```

Assume we have the following tuples for S1.1.4:
(‘S1.1.4’, 1, (1,1), t90)
(‘S1.1.4’, 1, (2), t91)
(‘S1.1.4’, 2, (1), t92)
…

We will output the following additional tuples:
sub(t103, t90)
sub(t104, t91)

We use the form sub(x,y) because when being used, a suitable parameter only needs to be the subtype of what is originally defined. If we output eq(x,y) in this case, we may create more constraints than needed, hence eliminate possible right solutions.

4.3.4 Base Constraints

In the next step, we generate some basic constraints that will be used later on as our starting point. With the aim of generating most of the basic constraints, we pay special attention to the rules’ conditions.

Basic types that we frequently encounter are:

- Number: integer, double
- String
- Boolean
- Other types

We have:

'S1.5.3'
no-main-role-active(user) <-
count-agent-activations(n,user),count-spine-clinician-activations(n,user),count-spine-admin-activations(n,user),count-patient-activations(n,user),count-third-party-activations(n,user),**n = 0**

Base on the constraint **n=0**, we can conclude that type of n is the same as type of 0 whose type is integer.

Based on what we know, we will output tuples such as:
base(t101, int)
base(t103, int)

…

In general, all count(x) where x can stand for any variable will also return an integer.

Constraints such as srv = "Spine", ra = "RA-ADB" will yield tuples where we have string as the type of t_n.
base(t20, string)

4.3.5 Selection Constraints

Among the rules' constraints that we encounter, sometimes we run across a constraint like the one below:

(S4.2.3)
canDeactivate(pat, x, Conceal-request(what, whom, start, end)) <-
hasActivated(pat, Patient()),
pi7_1(what) = pat

This constraint clearly states the relationship between variable what and pat. what is a set that contains seven elements whose first element is pat. To translate this kind of constraints into tuples, we will use the following form:

Let type of pat is t100, type of what is t102, we have:
select1(t100,t102): t100 is the first element of t102.

4.3.6 Constructor Constraints

We also want to focus on what type can a particular constructor return. Is it fine to replace a variable whose type is t1 with a function canDeactivate that takes t2,t3,t4 as its input? What's the relation between t1,t2,t3, and t4?

We will assign a return type to each function and atoms that we encounter.

(S4.2.3)
canDeactivate(pat, x, Conceal-request(what, whom, start, end)) <-
hasActivated(pat, Patient()),
pi7_1(what) = pat

We will have:
con(t1, Conceal-request, t102, t103, t104, t105))

…

Now that we already construct our database of constraints, we can move on with the second main step of the system – solving constraints to indentify variables' types.

4.4 Solve Constraints

In order to solve constraints, we have to use several rules to combine the existing constraints together. We will do this until there is no more combination left to solve. Although this method can help us to cover all cases, it will result in heavy memory usage and prolonged execution time. We will discuss about this matter and how to solve it in the next few parts.

4.4.1 Inference Rules

Through inferencing, we can introduce new constraints into our database and hence, create clearer views on the role and information of a particular variable. When we keep on doing this, eventually we will arrive to the point that we solve all the existing constraints and there is no new combination for us to consider. When we reach this point, we can stop the inferencing and start to analyze the result.

eq(x,y), base(x,t) -> base(y,t)

eq(x,y), con(x,C,y1,...,yk) -> con(y,C,y1,...,yk)

eq(x,y), tuple(x,y1,...,yk) -> tuple(y,y1,...,yk) one for each arity k

sel(x,y), tuple(y,y1,...,yk) -> eq(x,yi) one for each k for each i <=k

eq(x,y), eq(y,z) -> eq(x,z)

sub(x,y) -> eq(x,y)

4.4.2 Datalog Rules:

base(y,t) :- eq(x,y), base(x,t).

con(y,C) :- eq(x,y), con(x,C).

```
tuple(y,y1,y2,y3,y4,y5,y6,y7) :- eq(x,y), tuple(x,y1,y2,y3,y4,y5,y6,y7).

eq(x,y1) :- select1(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y2) :- select2(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y3) :- select3(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y4) :- select4(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y5) :- select5(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y6) :- select6(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).
eq(x,y7) :- select7(x,y), tuple(y,y1,y2,y3,y4,y5,y6,y7).

eq(y,x) :- eq(x,y).
eq(x,z) :- eq(x,y), eq(y,z).

eq(x,y) :- sub(x,y).
```

This part of the program was automatically generated according to [33].

Chapter 5

Datalog with Equality Constraints

5.1 Performance Problems

One of the problems we encounter when solving the constraints is the amount of time that we have to spend to run the program. Along the way, we generate:

- 3083 sub constraints
- 1656 con constraints
- 2161 eq constraints
- 7 sel constraints
- 181 base constraints

The combination of eq constraints together with sub and con constraints takes the toll on the performance of the analysis.

Constraints	Time
All constraints	Run more than 6 - 8 hours with no error produced
All but eq constraints	Instant
All but sub contrains	Instant
All + random 150 eq constraints	3-5 seconds
All + random 210 eq constraints (40 constraints more than above)	15-17 seconds
All + random 290 eq constraints (80 constraints more than above)	2 minutes 30 seconds

Table 5.1: Experiments time

To solve this issue, we go through several algorithms in order to optimize the size of the input or shorten the execution time of the performance. At the end, we decide to use the Union-Find algorithm for its simple but effective approach.

5.2 Union-Find

Union-Find algorithm helps us to merge the constraints together through the use of three functions:

• Make set: makeset(x) will return a set with x being its own parent if x originally did not belong to any set.

• Find: find(x) will return x if x is its own parent, else we will return find(x.parent). The final result will be the ancestor node at the top of the hierarchy.

• Union:. In our case, the order of x,y is not important, so we assume that union(x,y) will make the ancestor of x become the parent of y's ancestor.

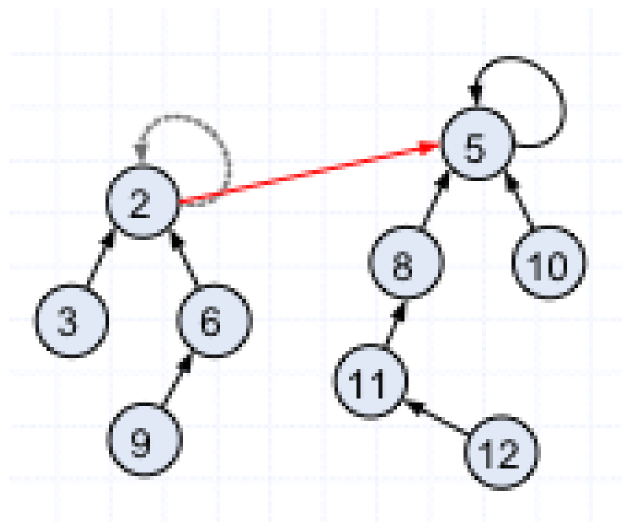

Figure 5.2: An Union - Find Example

Take this picture as an example, if we use find(x) for x being 3,8,6,or 2, we will get 2 back as the final result. If x takes on the value in {5, 8,10,11,12} we will get 5 back as the final result.

The red arrow in the picture symbolizes the command union(5,2), or to be more general, union(x,y) where x in {5,8,10,11,12} and y in {2,3,6,9}, according to our assumption about union(x,y).

With the help of Union-Find algorithm, we are able to optimize both storage/memory space and execution time. The program's respond time greatly improved.

We use the Union-Find code written by Josiah Carlson on August 12, 2003. Whenever an equal fact is considered, we applied the Union-Find algorithm to optimize the overall performance.

```
class UnionFind:
   def __init__(self):
      '''\
Create an empty Union-Find data structure.'''
      self.num_weights = {}
      self.parent_pointers = {}
      self.num_to_objects = {}
      self.objects_to_num = {}
      self.__repr__ = self.__str__
   def insert_objects(self, objects):
      '''\
Insert a sequence of objects into the structure.  All must be Python hashable.'''
      for object in objects:
         self.find(object);

   def find(self, object):
      '''\
Find the root of the set that an object is in.
If the object was not known, will make it known, and it becomes its own set.
Object must be Python hashable.'''
      if not object in self.objects_to_num:
         obj_num = len(self.objects_to_num)
         self.num_weights[obj_num] = 1
         self.objects_to_num[object] = obj_num
         self.num_to_objects[obj_num] = object
         self.parent_pointers[obj_num] = obj_num
         return object
      stk = [self.objects_to_num[object]]
      par = self.parent_pointers[stk[-1]]
      while par != stk[-1]:
         stk.append(par)
         par = self.parent_pointers[par]
      for i in stk:
         self.parent_pointers[i] = par
      return self.num_to_objects[par]
   def union(self, object1, object2):
      '''\
Combine the sets that contain the two objects given.
Both objects must be Python hashable.
If either or both objects are unknown, will make them known, and combine them.'''
      o1p = self.find(object1)
      o2p = self.find(object2)
      if o1p != o2p:
         on1 = self.objects_to_num[o1p]
         on2 = self.objects_to_num[o2p]
         w1 = self.num_weights[on1]
         w2 = self.num_weights[on2]
         if w1 < w2:
            o1p, o2p, on1, on2, w1, w2 = o2p, o1p, on2, on1, w2, w1
         self.num_weights[on1] = w1+w2
```

```
del self.num_weights[on2]
self.parent_pointers[on2] = on1
```

5.3 Use of Union-Find

Due to the lack of base constraints, we were not able to achieve the best result. Also, not all types will be solvable by this approach. We are able to group and divide them into groups where all the elements in the group will share the same type. If we know the type of one element in the group, we can conclude about all others.

We have 5491 variables (t1… t5491) in total. Among those, 3689 have types, or are grouped together according to their constraints on types. Types can be boolean, int, string, or ArgInt. (ArgInt comes from calls such as count(x) and other functions who return an integer when being called).

There are 2 major fixes that we have to insert into the generated codes in order to properly execute the program. With just the generated codes from [33], as we mentioned above, the program run into major performance issues. The first fix we did was to integrate Union-Find algorithm into the codes to reduce the running time. However, the program still takes a lot of time to respond, and the results were hard to analyze and read through. We got 4216 types results in total, but some of them were inconsistent with the type info.

The second fix gives us simpler, better results, and an overall boost in respond time. Though the results were reduced to 3689, they offered more information than the previous sets of results. The correctness and completeness of the analysis also improve a lot.

```
def replace(W,R,x,y):
  '''in W, replace all occurences of x with y;
     in R, replace facts that contain b with facts that contain a,
          and move them to W'''
  Wmatch = findfacts(W,x)
  W = W - Wmatch
  W = W | changefacts(Wmatch,x,y)
  Rmatch = findfacts(R,x)
  R = R - Rmatch
  W = W | changefacts(Rmatch,x,y)
  return (W,R)
```

and

```
rep = uf.find(fact[1][0])
```

```
if rep == x:
  (W,R) = replace(W,R,y,x)
elif rep == y:
  (W,R) = replace(W,R,x,y)
else:
  (W,R) = replace(W,R,x,rep)
  (W,R) = replace(W,R,y,rep)
```

5.4 Summary of Results Analysis

There are 3689 types in 349 union-find groups
Total number of types: 5491

Format of the outputs:
---- type_name : set of types, equivalent types

Example 1:

---- t1019 : set([('eq', ('t1005', 't1005'))]),
---- t1005 : set([('eq', ('t1005', 't1005'))]), t1019,

We can see that t1005 is the main types in the Union-Find set of [t1005, t1019]. We know that they are of the same types, but due to the lack of base constraints, we did not know if they are str, or int or other primitive types.

Example 2:

---- t5339 : set([('con', ('t5339', 'hasActivated', '**t1285**', '**t1285**', 't4125'))]),
---- t4753 : set([('con', ('t4753', 'hasActivated', '**t1285**', '**t1285**', 't3768'))]),
---- t4752 : set([('con', ('t4752', 'hasActivated', '**t1285**', '**t1285**', '**t1285**', '**t1285**', 't3736'))]),

We observe that **t1285** appears a lot in the results, so if we want to know more about a type information, we need to search through the table and locate **t1285**:

---- t1285 : set([('eq', ('t1285', 't1285')), ('con', ('t1285', 'Treating-clinician', 't1285', 't1285', 't3736')), ('base', ('t1285', 'str')), ('con', ('t1285', 'Agent', 't1285')), ('select1', ('t1285', 't957')), ('select1', ('t1285', 't969')), ('select1', ('t1285', 't2751')), ('con', ('t1285', 'Third-party')), ('con', ('t1285', 'NHS-health-org-cert', 't1285', 't3132', 't149')), ('con', ('t1285', 'Registration-authority', 't1285', 't3132', 't149', 't1602', 't3852')), ('con', ('t1285', 'Patient')), ('con', ('t1285', 'NHS-clinician-cert', 't1285', 't1285', 't1285', 't1285', 't4312')), ('select1', ('t1285', 't985')), ('con', ('t1285', 'Register-patient', 't1285')), ('con', ('t1285', 'Spine-clinician', 't1285', 't1285', 't3736')), ('con', ('t1285', 'Request-third-party-consent', 't1285', 't1285', 't3555')), ('con', ('t1285', 'General-practitioner', 't1285')), ('sub', ('t1285', 't1285')), ('select1', ('t1285', 't2763')), ('tuple4', ('t1285', 't1004', 't1005', 't1020', 't1021')), ('tuple7', ('t1285', 't1119', 't1102', 't1103', 't1104', 't1105', 't1106', 't1107')), ('con', ('t1285', 'Registration-authority')), ('con', ('t1285', 'Workgroup-member', 't1285', 't1285', 't3736')), ('base',

('t1285', 'int'))]), t3480, t3481, t3483, t776, t777, t775, t772, t773, t771, t778, t779, t174, t3001, t3002, t3003, t3556, t171, t172, t3009, t3558, t2570, t178, t179, t3823, t3820, t3826, t1958, t2682, t2685, t2684, t1950, t610, t1952, t1953, t1954, t1957, t2371, t2370, t2373, t2372, t2375, t2374, t2376, t2878, t3178, t3179, t3171…….

We can observe some inconsistencies in the above result since t1285 can be "str" and "int" at the same time.

Results with element names and rule numbers

Example 1:

---- n_S1.5.3_t228 : set([('eq', ('n_S1.5.3_t226', 'n_S1.5.3_t226')), ('base', ('n_S1.5.3_t226', 'int')), ('sub', ('n_S1.5.3_t226', 'n_S1.5.3_t226'))]), n_S1.5.3_t224, n_S1.5.3_t222, n_S1.5.3_t220, count<u>_S1.4.5_t3491, n_S1.5.3_t230, count<u>_S1.1.4_t3448, count<u>_S1.2.4_t3456, count<u>_S1.3.4_t3472, count<u>_S2.2.13_t3563, 0_S1.5.3_t231,

Example 2:

---- Ext-treating-clinician(pat,ra,org,spcty)_A5.3.5_t4244 : Ext-treating-clinician(pat,ra,org,spcty)_A2.2.1_t3994, Ext-treating-clinician(pat,ra,org,spcty)_A2.2.2_t3998, set([('eq', ('Ext-treating-clinician(pat,ra,org,spcty)_A2.2.5_t4007', 'Ext-treating-clinician(pat,ra,org,spcty)_A2.2.5_t4007')), (**'con', ('Ext-treating-clinician(pat,ra,org,spcty)_A2.2.5_t4007', 'Ext-treating-clinician', 'spcty_A2.3.7_t2146', 'spcty_A2.3.7_t2146', 'spcty_A2.3.7_t2146', 'spcty_A2.3.7_t2146'**)), ('sub', ('Ext-treating-clinician(pat,ra,org,spcty)_A2.2.5_t4007', 'Ext-treating-clinician(pat,ra,org,spcty)_A2.2.5_t4007'))]), Ext-treating-clinician(pat,ra,org,spcty)_A5.1.2_t4218,

Example 3:

---- NHS-health-org-cert(org,start,end)_S1.4.7_t3496 : spcty_A2.1.11_t1998, cli2_A4.1.3_t2652, spcty_A2.1.11_t1991, spcty2_A3.5.4_t2492, y_S2.4.8_t633, ag_P2.2.2_t1571, cg_A3.7.3_t2579, spcty_A1.1.3_t1626, pat_S2.4.4_t587, mgr_R1.1.3_t3086, mgr_R1.1.3_t3088, pat_S5.2.2_t1242…

Chapter 6

Implementation

The program is written in Python 2.5. It uses a developed parser written based on Toy Parser Generator to parse the policy input.

The program's length is 1670 lines, it can be broke down as following:
_ Policy Rule Definition Analysis: 200 lines of codes.
_ Generate constraints: 450 lines of codes.
_ Union-Find: 100 lines of codes.
_ Solve the constraints: Generated from [33], 700 lines of codes.
_ To put everything together: 250 lines of codes

The input it reads in are file of EHR policy, written in Datalog. In this project, we reads a total of four files: ra, spine, pds, hospital.

The program was run on a computer of:
_ Windows XP, Service Pack 3.
_ Duo-Core 2.0 Ghz
_ 2GB of RAM

Chapter 7

Conclusion

Constructing constraints plays an important part in our policy analyses. Constraints are the main keys we need to have in order to further examine the policy rules in details. They are used to express the type and subtype relationships between policy elements. Constraints are a means for solving type-checking, type-inference problems such as in our analysis. Constraints can have many forms to suit with different cases; e.g., we can express equality between two types, proper subtype relations, and when one type takes another type as a parameter.

A type analysis can be divided into two stages. The first stage's main goal is to collect and construct appropriate constraints. The second stage is to find solutions based on the data that are garnered in the first stage.

In this project, by doing a type analysis on the formal EHR policy, we can reinforce the results we have in our policy rule definition analysis, and also are able to see the policy rules with more type properties attached. While the policy rule definition analysis gives us a general view on how the policy works, and what main and sub divisions we can divide our classes into, type analysis helps us to realize what each class takes as its inputs, and the hierarchy between different variables in the policy.

Future work includes studies on how to achieve a more complete type analysis and its application to other aspects.

Bibliography

[01] Moritz Y. Becker, Cassandra: Flexible Trust Management and its Application to Electronic Health Records, University of Cambridge, October 2005

[02] Moritz Y. Becker, Andrew D. Gordon, and Cédric Fournet, SecPAL: Design and Semantics of a Decentralized Authorization Language, no. MSR-TR-2006-120, September 2006

[03] Moritz Y. Becker, SecPAL Formalization and Extensions, no. MSR-TR-2009-127, September 2009

[04] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T. SPKI certificate theory - IETF RFC 2693, September 1999.

[05] Juri Luca De Coi, Daniel Olmedilla. A review of trust management, security and privacy policy language. University of Hannover, 2008.

[06] Kagal, L., Finin, T. W., and Joshi, A. A policy language for a pervasive computing environment. In 4th IEEE International Workshop on Policies for Distributed Systems and Networks. 2003.

[07] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid, Access control meets public key infrastructure, or: Assigning roles to strangers, in Proceedings of IEEE Symposium on Security and Pri vacy, pp. 2-14, Apr. 2000.

[08] John DeTreville. Binder, a logic-based security language. In IEEE Symposium on Security and Privacy. 2002.

[09] P. Bonatti and P. Samarati. Regulating Service Access and Information Release on the Web. In Conference on Computer and Communications Security (CCS'00), Athens, November 2000.

[10] M. Dekker and S. Etalle. Audit-based access control for electronic health records. In Proceedings of the Second International Workshop on Views on Designing Complex Architectures (VODCA), pages 221–236, Amsterdam, September 2006.

[11] Qingfeng He. Requirements-based access control analysis and policy specificartion. North Carolina State University, 2005

[12] Butler Lampson, Martin Abadi, Michael Burrows, and EdwardWobber. Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems, 1992.

[13] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In CSFW, 2003.

[14] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role based trust management framework. In Proceedings of the 2002 IEEE Symposium on Security and Privacy, 2002.

[15] Richard Hayton, Jean Bacon, and Ken Moody. OASIS: Access control in an open distributed environment. In Proceedings of the 1998 IEEE Symposium on Security and Privacy, 1998.

[16] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In IEEE Symposium on Security and Privacy, 1996.

[17] D. Bell and L. La Padula. Secure computer systems: unified exposition and Mul-tics interpretation. Technical Report MTR-2997, MITRE Corporation, July 1975.

[18] Hasan Nageeb Qunoo. Verifying Access Control Policies through Model Checking. University of Birmingham, 2007.

[19] Bodei, C., Degano, P., Nielson, F., and Nielson, H. R. Static analysis of processes for no read-up and no write-down. Lecture Notes in Computer Science 1578, 1999.

[20] Lampson, B. W. "Protection," Proc. fifth Princeton symposium on information sciences and systems. Princeton university, 1974.

[21] Ferraiolo, D. F., and Kuhn, D. R. Role based access control. In Proceedings of 15th National Computer Security Conference, 1992.

[22] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The ponder policy specification language. In Proc. Policy 2001: Workshop on Policies for Distributed Systems and Networks, 2001.

[23] Zhang, N. Generating Verified Access Control Policies through Model-Checking. PhD thesis, School of Computer Science, The University of Birmingham, 2005.

[24] Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M. Enterprise privacy authorization language (epal 1.2). Technical report, IBM. November 2003.

[25] Uszok, A., Bradshaw, J.M., Jeffers, R., Suri, N., Hayes, P.J., Breedy, M.R., Bunch, L., Johnson, M., Kulkarni, S., Lott, J. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks. 2003.

[26] Simon Godik, T.M.: Oasis extensible access control markup language (xacml). Technical report, OASIS. February, 2003.

[27] Yuri Gurevich and Itay Neeman, DKAL: Distributed-Knowledge Authorization Language, 21st IEEE Computer Security Foundations Symposium (CSF 2008), 149–162.

[28] Yuri Gurevich and Itay Neeman, DKAL 2 - A Simplified and Improved Authorization Language, http://research.microsoft.com/apps/pubs/default.aspx?id=79528.

[29] Adam J. Lee, Marianne Winslett, and Kenneth J. Perano. TrustBuilder2: A Reconfigurable Framework for Trust Negotiation. Proceedings of the International Conference on Trust Management (IFIPTM 2009), 2009.

[30] Anderson, A.H., An introduction to the Web Services Policy Language (WSPL). Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004.

[31] Rita Gavriloaie, Wolfgang Nejdl, Daniel Olmedilla, Kent E. Seamons, and Marianne Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In 1st European Semantic Web Symposium (ESWS 2004), volume 3053 of Lecture Notes in Computer Science. 2004.

[32] Trevor Jim. SD3: A Trust Management System with Certified Evaluation. IEEE Symposium on Security and Privacy, May 2001.

[33] Y. A. Liu and S. D. Stoller. From Datalog Rules to Efficient Programs with Time and Space Guarantees. ACM Transactions on Programming Languages and Systems (TOPLAS), 31(6)1-38, August 2009.

[34] P.A. Bonatti and D. Olmedilla. Semantic web policies: Where are we and what is still missing? Tutorial at 3rd European Semantic Web Conference (ESWC'06), 2006.