Reusable framework for analysing System Models

Niels Thykier

Kongens Lyngby 2013
IMM-PHD-2013-90

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

IMM-PHD: ISSN 0909-3192

Abstract

In this day and age, systems and infrastructures are becoming more complex. Analysing these manually for weaknesses against insiders have become a daunting task. Accordingly, there has been several attempts to describe these systems in a way to make them machine analysable.

However, the work so far seems to have been limited to testing the feasibility and usability of these models. There has been no visible progress in creating an extensible framework to facilitate future work in this area.

In this project, I will attempt create a framework around these models to assist future work on these models. Previous work in this area tells us that there is interest for both static analysis and dynamic analysis

Contents

List of Figures

Chapter 1

Introduction

This chapter will briefly cover the rationale for this project and the layout of this report.

1.1 Reason for this project

This project is inspired by previous work in this area. Probst et al. have done several articles on the usefulness of system models[6, 7, 2]. Lindø also provided a proof-of-concept static analysis tool for these models[4]. But despite of this, we are still without a common platform or framework to deal with even the trivial parts of system models.

The purpose of this project is create such a framework, so that others can prototype their ideas without having to "reinvent the wheel" first.

1.2 Report structure

The report has the following structure:

- The Introduction chapter (chapter 1) describes the rationale for the project and the report layout.
- In the Background chapter (chapter 2), I will cover a bit about what a system model is and why they are useful.
- The Problem analysis chapter (chapter 3) describes my analysis of the problem domain and the requirements for the resulting framework.
- In the Design chapter (chapter 4), I will describe my approach to implementing the framework and some desired changes to the model specification language.
- The Implementation chapter (chapter 5) will cover the implemented framework and the major changes to the model specification language.
- In the chapter, Using the framework, (chapter 6), I will present various examples on how to use the resulting framework.
- The Future work chapter (chapter 7) will present some of the areas where expansion are possible or where the result might need improvements.
- The Conclusion chapter (chapter 8) will summarise the outcome of this project.

Chapter 2

Background

This chapter will cover some of the background information about system models.

2.1 Why system models

When an organisation has been the victim of a cyber crime, the best way of finding the perpetrator is usually analysing the log files and available records. However, the log files involved are usually huge, which makes it very hard to find entries relevant to the attack. This especially true if the attacker was an insider, whose actions at the surface may be indistinguishable from a regular daily activity.

As an example, consider the very small organisation shown in figure 2.1 (figure borrowed from Probst). In this small organisation, there is a secret document on the computer in the “User Office”. A copy of this document was stolen from the building at some point.

To prevent a future incident, we would like to ammend the security protocols. In order to accomplish that we have to find out how someone was able to steal

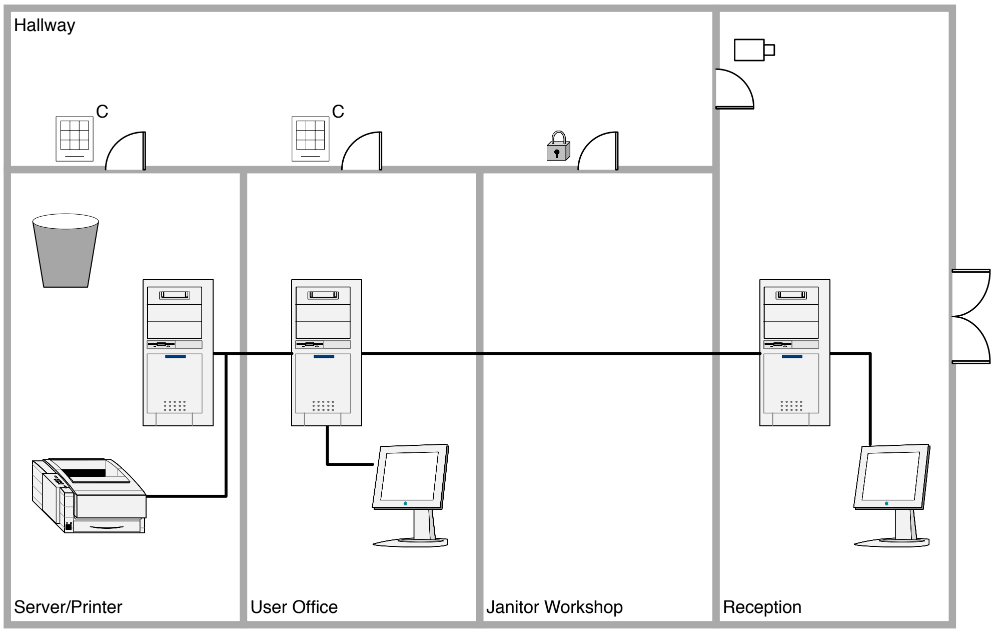

Figure 2.1: A small example organisation

the document in the first place. Assuming the thief might have been an insider, there are actually quite a few possible ways this could have happened.

It could have been so simple that the person working in the “User Office” just copied the document on to a USB stick, packed up his things and left like he does at the end of every work day. Or perhaps, he printed the document for proof-reading. Once done, he discarded the paper copy in the waste basket inside the server room. Here, the janitor noticed it when emptying the trash bin and figured it might be a good replacement for that bonus he never got.

To solve problems like this, Probst et al. proposed to describe organisations via “system models”[7] to make them easier to analyse and reason about. In layman’s terms, a system model is a mathematical model used to describe a company or an organisation. The model reasons about the structure of the organisation in multiple domains, including physical domain (i.e. the buildings and the rooms inside them) and the virtual domain (e.g. the interconnections between computers inside the organisation). An important aspect of a system model is that it also models access control rules. I will cover the more formal definition in section 3.3.

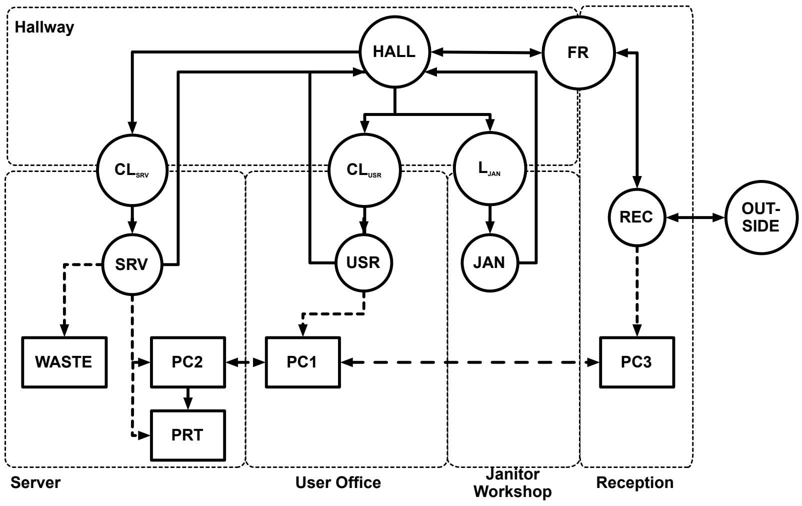

Figure 2.2: A graph representation of a small organisation

Given the sample organisation in figure 2.1, we might describe the organisation via the graph in figure 2.2 (figure borrowed from Probst). While the graph representation may assist us, it is still inadequate for automated analysis. It lacks information about things like access controls. Without these, we cannot reason about who could have done it (or who could not have done it). Based on Probst and Hansen's work[6], a system model of the same organisation could look like the one in figure 2.3 (this figure is also borrowed from Probst).

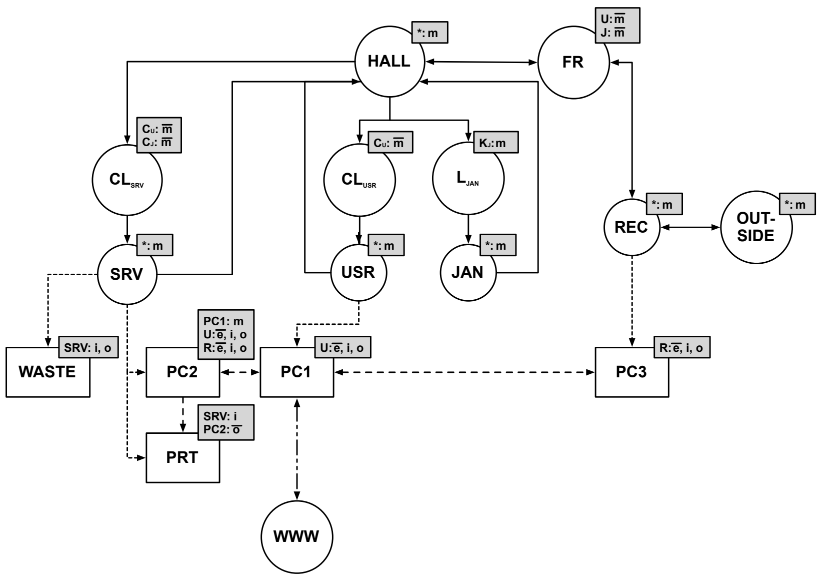

Figure 2.3: A representation of a system model for a small organisation

Chapter 3

Problem analysis

In order to develop any tool or framework, it is first necessary to figure out what problem it should solve. There are more than one way to approach this task, ranging from trivial “start with coding and see where it goes”-style to writing a full blown requirements specification. Inspired by “Simon Sinek” talk on “How great leaders inspire action”[8], I will try to start by answering the “Why”.

I strongly doubt that System Models have reached their full potential yet. As the world changes and we get better at using these models, new possibilities and ideas will appear. If these ideas are difficult to implement, some of them will die without never being tried. Therefore, it is important that we can easily create prototypes to test new ideas.

In the end, I am doing this project to make it easier to write new and extend existing tools (if any) related to system models.

3.1 Extensible

It is my goal to make it easy to write prototypes or tools to test new ideas around system models. The keyword here is "new" - namely, I want the result to be able to support almost any "crazy" new idea. To me that implies that I have to create something that can solve problems that I did not imagine existed.

If I cannot imagine the problems, how would I be able to make a tool that could solve them? Obviously, I cannot. But even if I cannot imagine all of the problems, I can still prepare a decent toolbox that is useful for solving a subset of them. In other words, I will be creating a framework or API for working with system models.

Like a chain being no stronger than its weakest link, so I believe that a tool (or framework) is no more extensible than its most rigid part. However, if you sacrifice everything in the name of extensibility, you will never get anywhere. The key to success is to pick just the right level of extensibility.

Finding that "right level" is possibly more of an art than a skill. Nevertheless, if this framework is to solve problems I cannot imagine, odds are that I cannot specify a final language to describe those problems. So the model specification should definitely be extensible, or failing that, at least replaceable.

To make it easier to write new tools, I intend to write an extensive framework with some replaceable parts.

3.2 Starting from what we know

I am not inventing an entirely new set of concepts; there is past work for me to start from. If I seriously want anyone to adopt my framework as the basis of their work, it must at least be able to solve their problems. A good starting point is therefore to look at dealing with the problems others had in the past.

System models have long been researched for the purpose of using them to perform "static analysis" of company (and its access controls). Probst and Hansen have written several articles on the topic for finding or protecting against "insiders"[7, 6, 2].

Probst and Hansen also worked on doing "online" analysis of a system [2]. This seems to be more of a dynamic analysis and probably including some form of simulation.

3.3 Definition of a system model

While the above tells us something about what system models have been used for (or what they do), it does not say much about what a system model is.

Probst et al. originally defined a system as $S =< I, Actors, Data, C, R, \Phi >$ (definition 6) and said it was a variation of a $\mu Klaim$ Calculus called "acKlaim". The individual items of the system tuple are[7]:

- I is defined as a directed graph (from definition 2).
- $Actors$ is a set of "actors" (from definition 3). These actors can perform actions in I[1].
- $Data$ is defined as a set of data items and can be stored on both nodes and actors (from definition 4)
- C is a mapping of an actor to a set of capabilities and R is a mapping from location or data to a set of restrictions. For each restriction r, the Φ_r function maps a capability to either "true" or "false" (from definition 5).
- Φ is then the set of all the Φ_r[2].

So, the gist of it. A system (model) is basically a directed graph with actors and data. There are restrictions on which actor can do what action (on data at a given node).

3.4 Analysis Conclusion

In my analysis, I have come to the conclusion that there will be new ideas, which extend existing research on system models. These extensions may likely

[1]The original definition says "move", but I took the liberty of assuming they could do more than that.

[2]As far as I can tell, it is not explicitly listed in any of the definitions. However, it appear to be used as such so I will go with this definition.

require changes to the code or the model specifications. To support these future extensions, I believe that either an extensible model specification language is needed, or alternatively the language needs to be replaceable.

From previous work in this area, I believe that the two major use-cases are static analysis and dynamic analysis of the models.

Chapter 4

Initial framework design

With the general knowledge of what this framework should focus on, I can continue with actually designing it. A major part of the framework will be the API exposed to tool writers. Hench, I believe that extra care and thought should be put into creating the actual API.

4.1 Good API design

Here I have been greatly inspired by Joshua Bloch, who did a talk on designing APIs. A couple of points, I took from his talk were (in no particular order):

- "Easy to evolve". Makes perfect sense for this framework considering I expect the use of system models to evolve.
- Spend more time on examples. People will "copy-paste" them and any bugs in the examples will end up in their code as well. [1, at 962s]
- Keep "concept complexity" of the API down.
- Code "against" the API early and often. [1, at 902s and 685s]

- Use a proper type instead of strings. If only strings are available, their format becomes the “defacto API”.[1, at 3270s]
- A good name makes it easier to explain. [1, at 1332s]

Bloch makes other recommendations; the list above were merely some I selected to focus on.

4.1.1 Avoid concept complexity

From my problem analysis, I concluded that I would need to support both static and dynamic analysis of the models. For me, these two types of analyses are two very different kinds. It is basically the difference between writing an optimising compiler and a program debugger.

Static analysis tries to reason about “all possible states” given only the input model. In contrast, dynamic analysis is based on not only the input model but also a concrete state.

If a programmer is only interested in doing (e.g) static analysis on the model, he/she should not have to worry about the dynamic analysis part (and vice versa). Based on this, it seem reasonable to make two distinct components for these two analyses. Since they will both work on a system model, it would probably be prudent to put that in a third component.

4.1.2 Coding against the API

The 3 component setup has another minor advantage. By implementing the component from “top to bottom”, I get to code against the “lower levels” of the API before writing it. This means I will naturally force myself to code against part of my own API.

Obviously, this is not a silver bullet. I got nothing depending on the “top” API itself. There may also be parts in the API that are not used by “higher levels”. Here I can and hope to compensate by writing examples uses.

As I understand Bloch, this is an iterative process. As such I cannot say what problems I will fix be using until I actually use it. But I definitely believe that I will not get everything right the first time. This technique will hopefully help me to shape the API and weed out design bugs as I discover them.

4.2 Language design

After having reviewed the existing language[7], I noticed a couple of things that bugged me:

- The language appears to have a lot of “boilerplate” syntax.
- Locations being a access granting credential.
- The example model have a fair share of nodes with the sole purpose of being an access control point.
- The language does not scale very well.

The boilerplate syntax mostly appeared in the form of having to explicitly state a lot of things, where omission or/and a good default would do. The rest of the issues will be covered in the following subsections.

4.2.1 Access rules

Analysing examples of the existing model specification language, I realised that all access rules were described on “nodes” rather than “edges”. While there is nothing wrong with that per se, formulating a concise API for them quickly became hard.

The original access rules specify 3 classes of credentials, which are:

- The actor doing the action.
- The location from which the action is being carried out.
- A credential which the actor possesses.

The “location” credential annoyed me quite a bit, because it effectively changes with every trivial move action. I suspected it could be an Achilles’s heel in the resulting API. Taking a step back I realised that a “location” credential could be described as an edge. Indeed, by pushing access rules to the edges of the graph I could remove the “location credential”.

A beneficial side-effect is that it removes some redundancy between the “locations” and “connections” sections. Consider the example model in figure 4.1.

```
// Small example model with a mistake
locations {
   office { *: m } (building);
   // Note: No access permitted from pc2.
   pc1 { U: elog, i, o; } (virtual);
   pc2 { pc1: m; U: elog, i, o; } (virtual);
}

connections {
   office -> pc1, pc2;
   pc1 -> pc2;
   // REDUNDANT/DEAD EDGE! pc1 does not allow any access
   // from pc2, so the edge does not serve any function.
   pc2 -> pc1;
}

// [... omitted...]
```

Figure 4.1: A small model with a minor mistake

This model contains a minor mistake in its access rule for its “pc1” location, so despite the fact there is a connection beween pc1 and pc2, the edge is completely useless. This model is based on the model by Probst and Hansen[6][p. 241], which seem to contain this very mistake. The full source of that model (with my corrections) is also included in figure 5.5.

On the other hand, if the access rules had been on the edges, I think it would be harder to accidentally declare “dead edges”.

From a language perspective, it would also make nodes used only to restrict access redundant.

This still left me with some issues.

- What happens with the access rules for accessing a document on the same node as the actor is on? In the old model specification, the node could have access rules like “U: i, o”.

- Documents / credentials on a given node can have access rules. These rules are “per document”, so they cannot be merged into the edge rules. Particularly, the specification from Probst and Hansen[6] implies that data actions can be logged.

- How do I solve these without forcing the programmer to check access controls “twice” (i.e. once on the edge and once on the node)?

At first I thought that the first problem was a non-issue. A basic assumption is that nobody will just leave their credentials or secret documents on the floor. Forcing the model writer to include a container (e.g. a safe or PC) for these cases seemed like the way to go.

However, what about “processes”? If somebody starts a process on a given PC, the assumption will be that it can (of course) read/write things from/to that PC. It is possible to work around this by either moving the data to an external device (e.g. adding a “hard disk” node) or by adding a “self-edge” in the model.

As for the access rules on documents. The “i” (read) and “o” (write) rules could probably just be merged with the access rule from the edge as needed. It would possibly complicate the implementation, but I believe it should be doable.

Encryption does not prevent people from obtaining the document; it prevents them from understanding its content. With that in mind, I think it would be reasonable to consider the decryption rules (i.e. “d”) a property of the document itself.

However, what about the “log decryption” action (i.e. “dlog”)? On one hand, it is unreasonable for “dlog” to be retained if I steal an encrypted document and decrypt it elsewhere (e.g. on my laptop outside the building). Instead it should be reduced to a regular decryption.

On the other hand, what if the decryption key is protected by higher credentials than I own, but I am permitted to access the decrypted document. This is quite common to be allowed to run “higher privileged” programs - e.g. using “su” or “sudo” on UNIX based systems. Indeed, there are real life examples of trying to restrict access to an encryption key without preventing employees from using it to do their job[3]. So the “dlog” access specifier might not be unreasonable in itself.

4.2.2 Large scale models

I do not believe the original language scales really well. Sure, you can use it for toy examples. But if you want to describe a complex company there are some fundamental places, where the model syntax falls short.

First, actors are described on an individual basis. Imagine a company of 200 employees spread across 5-6 departments, where every employee have their own ID card. That requires at least 400 lines (or statements) to create 200 employees + 200 unique keycards. That is without even considering that employees might have extra credentials based on the department they work in. If an employee is transferred to a new department (or promoted) you have to manually adjust all of his/her credentials.

Access rules have a similar issue. How do you specify that everyone in department A has access to room X by using their unique keycard. You cannot use a wildcard, because the other employees do not have access to the room. As far as I can tell, the only solution is to list *each and every permitted keycard*. Another rule you have to remember to update, when a new employee is added to (or removed from) department A.

It is impractical and a nightmare to keep up to date.

The best (but also the only) idea I had for solving this problem was to include "actor roles". The purpose of these roles is to describe "groups" of people in the model.

An example could be like this: "All employees have a keycard. Some Employees are Janitors, which also have a janitor key. Some employees are known as Researches." This could then be used to describe access rules like "the keycard of any Researcher" (which is stronger than "the keycard of any Employee"). This would then be combined with declaring actors as being "a Researcher", "a Janitor" or "an Employee".

These "roles" do not changes the expressive power of the languages. They are merely here to make larger models more concise and easier to maintain. Indeed, continuing with my 200 employee example from above; the model could be reduced to 200 lines to declare the employees plus a small number of statements for declaring the roles of the employees. Access rules could be similar reduced from having to list the keycards of all permitted employees to simply stating the keycard of a respective roles.

I have been trying to come up with a similar thing for the infrastructure graph, like describing a reusable component. However, I have yet to find the right level of abstraction for it. In particularly, all my attempts so far have lead to results resembling the start of a general purpose programming language. But at that point, people might as well just hand code the infrastructure directly in Java and be done with it.

4.3 Design conclusions

In my design, I have decided to follow some of Joshua Bloch's advise on how to design APIs. To reduce the concept complexity, I have decided to split the codebase into "components" based on their purpose.

I have identified some things in the model specification language that I would like to change. I would like to reduce the amount of boilerplate syntax and nodes only present to create access controls. Furthermore, I intend to introduce actor roles and actor role credentials to make it easier to maintain large scale models.

Chapter 5

Implementation

This section will cover what I have implemented. It will not cover the basic use cases (including some “consumer” APIs). For those, please see section 6, which has been dedicated for that purpose.

5.1 Notable changes to the language specification

Some of these changes have already been mentioned in the design section (see subsection 4.2). However, I also changed a few things in the language during the implementation phase. These should be seen in comparison to the specification used by Probst and Hansen[6].

Note that these sections do not cover changes caused by the implementation being incomplete. These, where deemed important enough to mention, will be covered in future works (e.g. subsection 7.4).

5.1.1 Reduced boilerplate syntax

The revised language features two basic optimisations to reduce the amount of boilerplate syntax.

First, many access rules can be omitted, which results in default values being used. For data policies, the default is simply “no restrictions”. For edge/connection policies, the default rule is “unrestricted move” (i.e. “*: m”). However, this default is only applicable if both nodes are of same domain and the domain is either “physical” or “virtual”. If these conditions are not met, the access rule for that connection must be specified.

Secondly, the locations section is now written in a more concise way. Previously, locations were specified via the syntax:

```
locations {
  nodeA { access-rules } (domainX)
  nodeB { access-rules } (domainX)
  nodeC { access-rules } (domainY)
}
```

Now, the same is now written as:

```
locations {
  domainX: [nodeA, nodeB],
  domainY: [nodeC]
}
```

The idea here is to remove the repeated domain specification. This pays off because there are only 4 domains, but there can be many nodes in any of those domains.

Note the missing access rules in the revised language have been moved to the connections section (as debated in subsection 4.2.1).

5.1.2 Credential types

All credentials now have one of the 3 following types:

- physical key ("key") - A physical credential (e.g. a key or an identity badge).
- pass phrase ("password") - A "secret" that is known by the owner (e.g. a pass phrase).
- identity - The identity of the actor. All actors always have exactly one instance of these, which is implicitly pre-declared.

The "key" and "password" credentials can have any name allowed by the syntax except for any keyword ("key", "locations" etc.) or the name "identity". The name, "identity", is exclusively reserved for the "identity" credential (which, incidentally, can only be referred to by that name).

The type specification can generally be omitted when the name is non-ambiguous. This means that most access rules will generally be syntactically unaffected by the introduction of types. Although, when disambiguation is needed, the credential name is prefixed with its type surrounded by "<>" (e.g. "name" becomes "<key>name" or "<password>name"). There is an example of this in figure 5.2.

Note that identical credentials are considered an inherent part of the actor and cannot be shared in any way. For most parts, they behave like an actor role credential as well as an actor credential (see subsection 5.1.4).

5.1.3 Actor roles

It is now possible to declare actor roles. While their use is entirely optional, I believe they will greatly reduce the maintenance burden of any non-trivial model. These roles are declared in their own section called "actor_roles", which (if present) must appear between the "connections" and "actors" sections.

Each role can extend previously defined roles and have 0 or more credentials associated with them. All credentials from super roles are inherited. An example "actor_roles" and "actors" section is shown in figure 5.1.

Note that actor role credentials can used both as an actor role credential and as an actor credential. Please see the next subsection for the details.

```
actor_roles {
    Employee {
        password employee_code;
    }
    Janitor extends Employee {
        key janitor_key;
    }
    SeniorJanitor extends Janitor {}
}
actors {
    U (Employee) @ outside;
    SJ (SeniorJanitor) @ outside;
}
```

Figure 5.1: Example of an actor_roles and actors section

5.1.4 Access rules using role or actor credentials

When the credential is associated with an actor or an actor role, the key name is prefixed with name of actor (or actor role) followed by a period (e.g. "name" becomes "actor.name" or "role.name").

A notable change from the Probst and Hansen is that an actor is no longer considered a valid credential[6]. These uses should generally be replaced with the identity credential of that actor (i.e. "actor" becomes "actor.identity").

Another change is caused by actor roles. All credentials related to roles can be used in any of 3 ways. First, the credential can be used with the role that declared it. Secondly, the credential can be used with any "sub-role" of that role. Finally, the credential can also be used with an actor possessing that role.

The difference between the two first is similar to that of class inheritance from Object-Oriented Programming languages. Going from "Set" to "SortedSet" is more specific and limits the possible choices for implementations. Similarly, going from "Janitor.janitor_key" to "SeniorJanitor.janitor_key" limits the credential to any "janitor_key" initially owned by a "SeniorJanitor" rather than a "Janitor" (reusing the example in figure 5.1). The third way of using these credentials, would be "SJ.janitor_key". This simply declares that only the "janitor_key" of the actor "SJ" can be used.

The "identity" credential is (once again) a special case. While it is an actor credential, it may be used as if it had been an actor role credential. In particular,

```
// "locations {...}" has been omitted
connections {
    n1 -> n2 {  <role>SJ.<key>c: m; }
    n2 -> n3 {  <actor>SJ.<password>c: m; }
    // Not ambiguous; just being explicit for the sake of it.
    n3 -> n4 {  <actor>J2.<key>k: m; }
}

actor_roles {
    J {
        password c;
        key c;
        key k;
    }
    // Here "J" is not ambiguous (can only be a role)
    SJ extends J {}
}
actors {
    J (J) @ n1;
    // Here, the "(J)" is not ambiguous (can only be a role).
    J2 (J) @ n1;
    SJ (SJ) @ n1;
}
```

Figure 5.2: Example showing how to disambiguate credentials in access rules

using “Janitor.identity” means that any actor having the “Janitor” role (possibly indirectly) can use his/her identity credential to satisfy this requirement.

The keen reader might have been wondering up to know “what happens if an actor role and an actor has the same name”. Technically, this is permitted. Where this makes references to credentials ambiguous, it is possible to disambiguate them by prefixing the actor or actor role name with “<actor>” or “<role>” (respectively). An example of the disambiguation syntax can be seen in figure 5.2.

Note that ambiguity is not allowed, even when either alternative would leave only one possible credential owned by the same actor.

```
// Other sections omitted
connections {
  n1 -> n2 {
            // GOOD: Syntactical and semantically valid example:
            *: i (1%, 2), // Annotation for "i" is "(%1, 2)"
               o (0%);    // Annotation for "o" is "(%1, 1)"
                          // (due to time-cost defaulting to 1).
         };

  n1 -> n3 {
            // Syntactical valid, but semantically INVALID example
            // (Just to show the string syntax)
            *: m ("hello world");  // INVALID (semantically)
         };
}
```

Figure 5.3: Example showing the syntax of annotations

5.1.5 Access rule annotations

Another addition to the language is the “access rule annotations”. These annotations provide some additional information about the access rule.

These annotations can be attached to the capabilities of an access rule enclosed in parentheses. The syntax allows the values of an annotation to be either an integer, a percentage or an arbitrary string. The syntax is shown in figure 5.3.

Currently, only two annotation values are defined. These are (in order) “detection risk” and “time-cost”. The “detection risk” is a percentage, that is an integer between 0 and 100 inclusive followed by a “%”. The “time-cost” value is simply a positive integer. If omitted, they default to “0%” and “1” (respectively).

As the name might imply, the framework considers these values as “comments” with no special meaning by default. However, it does provide access to these values, so consumers can apply any meaning they choose. See also subsection 6.4.

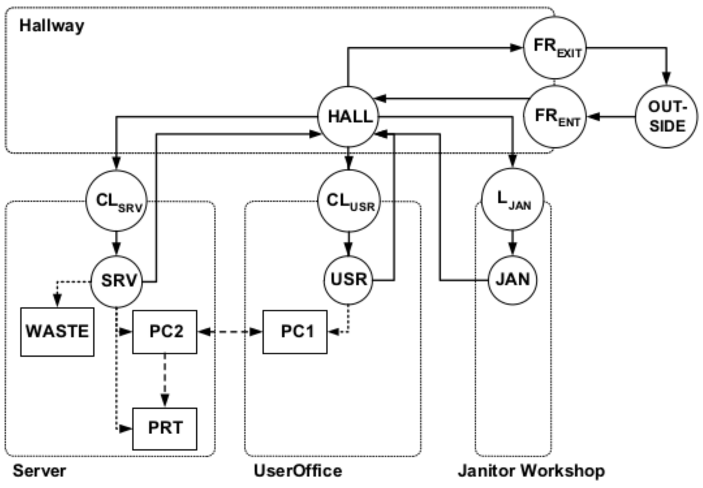

Figure 5.4: Visual representation of Probst and Hansen model

5.1.6 Putting it all together

It may be hard to fully appreciate all of these changes. In figure 5.4, you will find the example model specification from Probst and Hansen[6][p. 237]. That model corresponds to the textual representation in figure 5.6 (based on [6][p. 241]). I have added some inline comments to the model to clarify where I have made changes and what those changes were.

In contrast, figure 5.6 contains the same model rewritten in the revised language. In my rewrite, I have taken a couple of liberties. First, I have promoted all access controls to use actor role credentials. Assuming I have translated them exactly as Probst and Hansen had intended them, new agents can be added to the model without needing any changes to the infrastructure.

Secondly, a lot of access rules are now implicit and quite a few nodes have been removed. The old model has 14 nodes, while the rewritten model describes the same with only 9 nodes. Likewise, the number of edges have decreased from 19 to 12 (counting two-way edges as 2 separate edges).

Note that despite the fact that the parser can actually parse the rewritten variant of the model, the execute (i.e. "e" and "elog") actions are not actually supported (see subsection 7.1).

```
locations {
    outside {} (building);
    entry { U: mlog; J: mlog; } (building);
    exit  { U: mlog; J: mlog; } (building);
    hall  { *: m; } (building);
    lock_jan { key_jan: m; } (building);
    jan { *: m; } (building);
    lock_usr { code_U: m; } (building);
    // NB: original spec used "user" for this node
    // but it appears to be referred to as "usr" everywhere else.
    usr { *: m; } (building);
    // NB: Added "pc2: m;" since there is an "pc2->pc1"-edge
    // and it otherwise seem to serve no purpose (as "U" cannot
    // enter "pc2")
    pc1 { U: elog, i, o; pc2: m; } (virtual);
    // NB: fixed missing semi-colon in access rule (after code_J: mlog)
    lock_srv { code_U: mlog; code_J: mlog; } (building);
    // NB: original spec used "server" for this node
    // but it appears to be referred to as "srv" everywhere else.
    srv { *: m; } (building);
    pc2 { pc1: m; U: elog, i, o; } (virtual);
    printer { srv: i; pc2: olog; } (device);
    waste { srv: i, o; } (object);
}
connections {
    outside -> entry;
    entry -> hall; exit -> outside;
    hall -> lock_jan, lock_usr, lock_srv, exit;
    lock_jan -> jan; jan -> hall;
    lock_usr -> usr; usr -> hall; usr -> pc1;
    pc1 -> pc2; pc2 -> pc1;
    lock_srv -> srv;
    srv -> hall, waste, pc2, printer;
}
actors {
    U @ outside;
    J @ outside;
}
data {
    code_U { } @ U;
    code_J { } @ J; key_jan { } @ J;
    // NB: Replaced an "r" with an "i" (DataActions does not have "r").
    secret_file { U: i } @ pc1;
}
```

Figure 5.5: The original model from Probst and Hansen

```
// Re-written standard example
locations {
    physical: [outside, hall, jan, usr, srv],
    virtual: [pc1, pc2],
    device: [printer],
    container: [waste],
}
connections {
    outside -> hall { Employee.employee_code: mlog; };
    hall -> outside,
            jan { Janitor.janitor_key: m; },
            usr { User.employee_code: m; },
            srv { Employee.employee_code: mlog; };
    usr ->  pc1 { User.identity: elog, i, o; },
            hall;
    pc1 ->  pc2;
    pc2 ->  pc1,
            printer { *: olog; };
    srv ->  waste   { *: i, o; },
            printer { *: i; },
            hall;
}
actor_roles {
    Employee {
        password employee_code;
    }
    User extends Employee {}
    Janitor extends Employee {
        key janitor_key;
    }
}
actors {
    U (User) @ outside;
    J (Janitor) @ outside;
}
documents {
    secret_file { User.identity: i; } @ pc1;
}
```

Figure 5.6: The model from Probst and Hansen rewritten in the next syntax

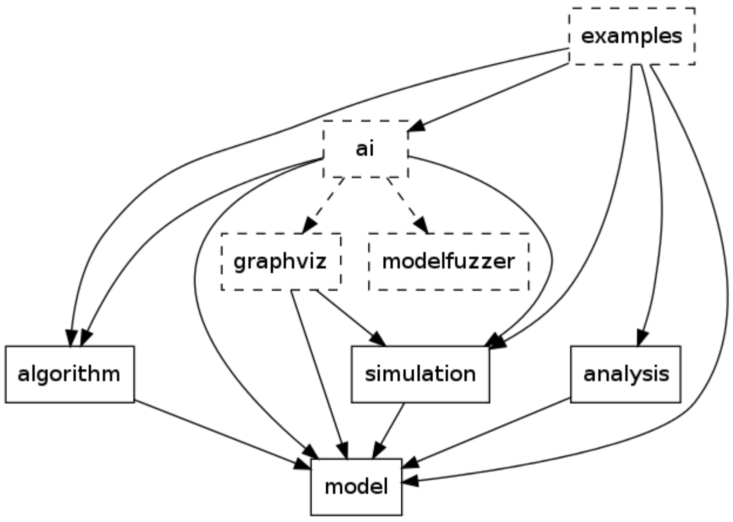

Figure 5.7: Overview of how the components relate. Solid boxes are main components, dashed boxes are non-API/consumers. Solid arrows represent strong dependencies, dashed arrows are dependencies only used during "testing".

5.2 Components of the framework

The framework is split into 4 "main" components. There are also a couple of extra components, which will briefly be mentioned in the end of this section.

Figure 5.7 provides a quick overview of how the components relate with each other. Each box in that figure represents a compontent. Solid boxes are part of the "main" framework and the dashed boxes are "other" components (see subsection 5.2.5). A solid arrow represent that source node has a strong dependency on the target node. The dashed arrows are dependencies only used during testing of the component.

5.2.1 "model" component

The "model" component is the main component. It contains the most basic parts of the framework including the representation of the model as well as the model parser.

Currently, the only way of obtaining a `SystemModel` is by parsing a specifica-

```
connections {
     hallway -> office        // Edge (source/dest)
         { key:               // Requirement 1
                mlog (0%, 2)  // Action, log info plus risk/time annotation 1
           *:                 // Requirement 2
                m (10%, 5)    // Action, plus risk/time annotation 2
         };
}
```

Figure 5.8: Example of an access policy in the language

tion using a `SystemModelParser`. The `SystemModel` itself is implemented as a `Graph`, where the nodes (`SystemGraphNode`) stores documents and credentials while the edges (`SystemGraphEdge`) contains the access policies.

In the language specification, an access policy is described as requirement that maps to capabilities (see 5.8). But when parsed, these access rules are remapped in quite a different way. Namely, edges map a desired capability to an `SystemAccessPolicy`, which consists of 1 or more `SystemAccessPolicyComponents`. These components then contain information about credential requirements, logging and the annotations (the latter being described in 5.1.5).

The rationale for this consists of several reasons. For one, I felt that a set of requirements made a really poor choice for the key in a key-value mapping. Especially with actor roles, it could be really difficult for consumers to "construct" these keys, which I felt would make them useless as a key.

Secondly, I believed that most consumers would generally be more interested in what capabilities were possible and then in what were required for those capabilities. In a more informal tone, I prioritised the "I want to do X, what does that require?" use case over the "I have the set of credentials S, what does that allow me to do?" use case.

So to describe the re-mapped setup as it looks in the framework using a "model language"-like syntax, it would be something like the pseudo-example in figure 5.9.

Log rules Another change between the language and the framework is that the framework allows more fine-grained rules about that is logged. All logged capabilities (e.g. "mlog") are simply mapped to a pre-defined set of logging

```
/* PSEUDO example of how access rules are structured in the framework.
 * NB:  This uses a "pythonesque" syntax, so {x:y, z:a} is a
 *      mapping from x to y and from z to a.  [x,y] is a
 *      list consistent of the elements x and y. set([x, y]) is
 *      a set consisting of the elements x and y.  Special-case,
 *      set() is the empty set.
 */
connections {
     hallway -> office                      // Edge (source/dest)
         { m:                               // Action
             set([                          // Set of components
               { "restrictions": set([key]), // Restrictions 1 (component)
                 "annotation": (0%, 2),      // Annotations  1 (component)
                 "log-rules: set([...]),     // Non-empty set of Log rules
               },
               { "restrictions": set(),      // Restrictions 2 (component)
                 "annotation": (10%, 5),     // Annotations  2 (component)
                 "log-rules: set(),          // Empty set of log rules
               },
             ]),
         };
}
```

Figure 5.9: Pseudo-example of an access policy in the framework

rules - these are marked with a * in the list below. The implemented logging rules are:

- action taken * - The action that triggered the log entry (e.g. "move")
- source node * - The source node of the actor performing the action.
- target node * - The target node of the action performed by the actor (if any).
- possessible * - The possessible involved (e.g. being read) in the action (if any).
- all-credentials * - All the credentials used to authorise this action (if any).
- the actor - The actor performing the action.

The "model" component also contains a number of auxiliary classes for representing and creating "traces" and "log files". At first, I thought they might be better suited for the "simulation" component. However, on second thought I realised it could make sense for an analysis to use those utilities to represent its findings.

5.2.2 "algorithm" component

The "algorithm" component contains a few utilities related to path finding. It has two different types of path finders. The first is a standard "find a path"-path finder. The other "path finder" will find all simple paths from A to B. The former is called `PathFinder` and the latter a `PathLister` (in the lack of a better name for it).

The `PathFinder` was implemented as a "breadth-first" path finder, so it always finds the shortest path between the two nodes. The `PathLister` uses a "depth-first" approach to finding all the simple paths. These implementations are hidden away and their capabilities are simply documented as a part of the API.

These path finding tools work on models as simple directed graphs. Even if the `SystemModel` interface revised, path finders will not need to be revised as long as the revised model interface is (or contains) a `Graph`. To make this possible, some decision logic has deferred to `Predicates`. These predicates are used to determine whether a given edge be crossed or not.

The component also includes a number of utilities related to path finding and the resulting paths. It is quite possible that eventually the number of these utilities will grow to better reflect the future use-patterns.

5.2.3 "analysis" component

The "analysis" component is intended for static analysis. It features a fixpoint analyser and an abstract fixed-point analysis.

The abstract fixpoint analysis provides a basic implementation to facilitate writing of new analyses. It expands a basic interface and solves a lot of the ground work. This include things like associating nodes with data from the analysis and keeping track of which nodes have been analysed (successfully).

As with the path finding, these analysis tools currently work with models as directed graphs. So they are also reusable on a revised `SystemModel` interface as long as it remains (or contains) a graph. Of course, individual analyses that rely on the current semantics of the models would still be affected. But the effects would hopefully be limited.

5.2.4 "simulation" component

The "simulation" component concerns itself with analysing the models via AIs or Jung agents. Its main purpose is to provide the `SystemModelSimulator` as well as an interface between the simulator and the AIs.

The simulator itself will maintain a `SystemModelTrace` and `SystemModelLog` of all actions occurred in the simulator. The trace records all actions that occurred, as if there had been perfect surveillance of all actors. The log will only contain the subset of all traced actions that are actually logged. Furthermore, entries of the log file will omit details based on the logging rules (described earlier in 5.2.1). Thus, if every action is vigorously logged down to every last detail, the trace and the log will basically contain the same information.

The simulator also provides support for event listeners. The events emitted by the simulator are mostly in a 1:1 correspondence with the traced actions. Though there are a few simulator events that describes "meta" changes to the state of the simulator (e.g. the "start of simulation" event).

In the simulation, the ordering between the actions of any actors are not actually defined. This in turn provides a source of non-determinism. In the simplest case, assume two agents ("A" and "B") reach for the same credential. In this simple situation, there are 3 possible resolutions.

1. "A" succeeds in taking the credential and the action of "B" fails.
2. "B" succeeds in taking the credential and the action of "A" fails.
3. Both agents fail to carry out their actions.

When such non-determinism is detected, the simulator defers the problem to the a conflict handler (`SimulatorActionConflictHandler`). The conflict handler can then resolve the problem by choosing an ordering of the actions and (at its discretion) unconditionally fail any actions of its choosing.

Once the problem has been resolved by the conflict handler, the simulator plays out the "surviving" actions in the order chosen. Note that the simulator may still fail some of the surviving actions, if it turns out the actions cannot be carried out anyway. This has the benefit of avoiding unnecessary logic in the conflict handler (e.g. for checking whether actions would eventually succeed or not).

5.2.5 Other (non API) components

There are a few other very small components in the source tree of the framework. These are not actually part of the framework as much as they are example consumers or extensions of the framework.

- ai - This component contains a few AI/Jung agents implemented by Emil Gurevitch
- modelfuzzer - This component contains a tool to auto-generate random models for "fuzz" testing. This was also implemented by Emil Gurevitch. Since it is written in mostly python, it is the only component that does not depend on any of the "main" components.
- graphviz - A component by Emil Gurevitch to transform simulated models to DOT graphs.
- examples - A component containing a small set of example code snippets demonstrating how to use the framework.

5.3 The model parser

The default implementation of the `SystemModelParser` is split into 3 parts. Each part handles one or more "phases" in the parsing. The first two parts are basically (and unsurprisingly) a lexer and an abstract syntax tree (AST) generator. These two parts are auto-generated from an ANTLR specification. Finally, the last part handles checking of the semantics and constructing a `SystemModel`.

Originally, I had envisioned a two-phase parsing, where the AST generation was skipped. However, in the end I realised including the AST step was a better idea. First of all, interleaving Java code into the parser-specification greatly increased the complexity of many of the grammar rules. Especially, if the input context (e.g. line number of the input) were to be retained. On the other hand, the AST generated by ANTLR included line information without any extra work.

The second major problem was producing useful error messages. Personally, I am not too happy with some of the default error messages in the generated parser. I solved most (but not all) of that by making the lexer more lenient than it ought to be. As an example, the lexer accepts any valid "name" (e.g. "ingolf") in place of a capability (like "m" or "mlog"). Invalid capabilities are then rejected during the third phase, where the semantics are verified.

This may seem like a lot of extra work at first, but it meant that I could replace a lot of ANTLR's "no viable alternative" error messages with a more human readable one. Sadly, it is still possible to trigger some of the "no viable alternative" error messages. This usually occurs when the punctuation is wrong (e.g. a "," that should been a ";").

Thanks to ANTLR's tree-rewrite support, it was possible to reduce the complexity of the generated AST. It also allowed the semantic checker to visit various parts of the tree in the order desired rather than in the input order. As an example, the checks of the "connections"-section are run last despite it being the second section parsed. At that time, the "credentials", "actors" and "actor_roles" sections have already been processed allowing the access rules to be checked in a single pass.

5.3.1 Adding new annotation

Another convenient side-effect is that adding support for a new annotation is quite trivial. It can even be done without regenerating the auto-generated parts of the parser. In the `ParserFrontend` class, there is an array annotation checkers

```
// PERCENTAGE and POSITIVE_INT is a Function<String,AnnotationValue>
// (i.e. they map a String to an AnnotationValue)

static final PolicyChecker[] ANNOTATION_CHECKS =
  new PolicyChecker[] {
    checkedAnnotationValue(
        SystemAccessPolicyComponent.ANNOTATION_KEY_DETECTION_RISK,
        "Detection risk must be a percentage ([...])",
        "0%", PERCENTAGE),
    checkedAnnotationValue(
        SystemAccessPolicyComponent.ANNOTATION_KEY_ACTION_TIME,
        "Time-cost must be an integer greater than 0",
         "1", POSITIVE_INT),
};
```

Figure 5.10: Annotation checkers in the `ParserFrontend`

(see figure 5.10).

To add a new annotation, one simply needs to add a new value to the end of that array. As seen in the example, this is done by calling `checkedAnnotationValue` with 4 arguments.

1. "key" name of the annotation. This is used by consumers to access the value.
2. A human readable description. This is the error message provided if the parsed annotation is not semantically valid.
3. A default value provided as a `String`. This will be passed to the converter function (the next argument) as-is.
4. A converter `Function` that maps a `String` to an `AnnotationValue`. If the input is not valid, the converter should simply return null.

With that, the parser frontend will handle the rest.

5.3.2 Auto-generated models

The model parser also supports two "special" comment tokens. These tokens can be used to change the parsers idea of what the current "file name" or "line number" is. The first is the "#line" directive used by many C/C++ compilers. The other is the line directive left behind by the GNU GCC Preprocessor. These look like this:

```
// 1. With null return
// - getNodeByName will return null if node is unknown
//   (Similar to Map.get(Object))
public @Nullable N getNodeByName(String name);

// 2. Without null return with "hasX"-test method
// - getNodeByName will throw an exception if node is unknown
// - hasNode will return true if node is known
public boolean hasNode(String name);
public N getNodeByName(String name)
    throws IllegalArgumentException;

// 3. With Optional
// - Returns Optional.absent() if node is unknown
public Optional<N> getNodeByName(String name);
```

Figure 5.11: Example of Optional, NonNull and Nullable APIs

```
#line 314 "some-other-file"
# 314 "some-other-file"
```

With these tokens, it is possible to create models from other “higher level” specifications and still have the model parser provide correct error information. Even if this “higher level” is simply just running it through a C-preprocessors to get “#include” or macros, which the current parser does not support.

5.4 Immutable and NonNull/Optional by default

Very early in the development, I remembered the two most common source of bugs in my own programs. These can basically be summed up as “null pointers” and “state”. When I adopted the Guava libraries, I revised my approach and my API to avoid these problems preemptively.

On the “null pointer” front, I plugged every part of the API to never return “null” pointers. This was possible due to the `Optional` from the Guava libraries and sometimes even made the API simpler. Figure 5.11 is an example of this based on the `getNodeByName` from the `Graph` API.

Similarly, I stopped allowing “null” arguments in all public parts of the API. In my experience, this greatly reduced the “cognitive load” of implementing the code. Internally, there are some places where “null” is used or even passed to or returned from methods. Examples include objects that are lazy loading some

```
Set<Object> mutable = new HashSet<>();
Set<Object> unmodifiableView = Collections.unmodifiableSet(mutable);
assert unmodifiableView.size() == 0;
mutable.add(new Object());
/* Holds because unmodifiableView is a view and therefore
 * indirectly mutable */
assert unmodifiableView.size() == 1;
```

Figure 5.12: Snippet showing that “unmodifiable” does not mean “immutable”

```
  // Without ImmutableSet
  /**
   * @return A {@link Set} of the credentials.  This set is
   * immutable and cannot be modified.  The only problem is
   * that you will forgot this fact in five minutes and will
   * have to review this piece of documentation to remind
   * yourself that is was not actually a view.
   */
  public Set<SystemCredential> getCredentials();

  // With ImmutableSet
  /** @return An {@link ImmutableSet} of the credentials.
   *   If you store this in variable with type "ImmutableSet",
   *   you can immediately see it is not a view.
   */
  public ImmutableSet<SystemCredential> getCredentials();
```

Figure 5.13: On returning unmodifiableSet vs ImmutableSet

of their fields.

The second problem, “state”, is a bit harder to deal with (at least in an imperative language like Java). For most parts, I had been trying to solve this by using things like `unmodifiableSet`. But here, the Guava libraries came to my aid again with its `ImmutableCollections`.

To the untrained, this may not be obvious but the `Set` returned by `unmodifiableSet` can still mutate. Figure 5.12 features a small snippet demonstrating how to do this.

Accordingly, in my API I had to add a lot of boilerplate comments about whether a given set could be a view or not. With an `ImmutableCollection` this problem disappears, because it is an immutable copy of the original collection. A lot of those boilerplate comments could go away by promoting types (see the example in figure 5.13).

5.5 Static factories vs Constructors

Throughout the framework, I have strongly preferred static factories to “public” constructors in all of the API. My rationale is that it allows the underlying implementation to be replaced without affecting any consumers.

During the development of the framework, I exploited this to replace the underlying implementation of several classes. Even now, these static factories provides a mean to change the default implementation for many common classes like path finding.

Another bonus of this is that allows the static factory to optimise some special cases, like giving a different type of object depending on the arguments. When dealing with immutable objects, such optimisations are actually fairly trivial to exploit.

As an example of these optimisations, the current implementation of the “Breadth-first” path finder happens to be “reentrant”. Thus, the static factories currently provide the path finder as a singleton object.

That said, when the framework is more mature, it might make sense to expose some of these implementations and their constructors in the API to allow subclassing.

5.6 Known issue: inline Predicate (etc.) are messy

Currently, the use of `Predicates`, `Functions` and the likes are a bit “heavy” on the syntactical side. This problem is not limited to this framework per se. However, since the framework promotes the use of predicate functions in its API, users of the API are likely to be affected by it.

The problem is a consequence of the Java syntax itself. Java insists on predicate functions being declared as a full class (even if anonymous). This causes a one-line predicate to result in about 4-5 lines of Java boilerplate code.

Allegedly, Java 8 will have better support for this via its “Lambda” enhancements[5]. The (expected) reduction in boilerplate syntax can be seen in figure 5.14.

```
// Without lambda support
final Set<SystemCredential> creds = ...;
new Predicate<SystemGraphEdge>() {
    @Override
    public boolean apply(@Nullable SystemGraphEdge e) {
      return Objects.requireNonNull(e)
                 .hasSufficientCredentials(t, creds);
    }
};

// With the (proposed) lambda support
final Set<SystemCredential> creds = ...;
(e) -> Objects.requireNonNull(e).hasSufficientCredentials(t, creds);
```

Figure 5.14: Anonymous classes vs. lambda expressions

5.7 Required libraries and platform

The framework is written entirely in Java 7 (using OpenJDK-7). I also used the "Null-Analysis" annotations from Eclipse (org.eclipse.jdt.annotation) to assist with find possible null pointer issues. The framework also exposes some of the classes from the Guava libraries in the API.

Furthermore, there are a couple of extra requirements, which are not visually exposed in the API. The parser implementation uses ANTLR 3.2 and the AIs implemented by Emil Gurevitch uses Log4J. Finally, JUnit4 was used for testing.

5.8 Implementation conclusion

The revised language has quite a few changes, including actor roles, typed credentials and annotations.

The implemented framework features 4 major components, which are called "model", "algorithm", "analysis" and "simulation". The latter 3 component are generally dedicated to path finding, static analysis and dynamic analysis/simulation (respectively). The "model" component provides most of the underlying API needed or shared by the other components.

The API exposed by the framework generally prefers immutable objects and disallows null pointers in ever place where it is feasible.

CHAPTER 6

Using the framework

This section contains examples of how to accomplish various tasks with the framework.

6.1 Basic usage of the framework

In this subsection, I will briefly cover some of the basics of how this framework can be used. It involves mostly use-cases and example solutions to them.

6.1.1 Parsing models

Almost any task using this framework will start with the need for parsing a model. This is especially true since there is currently no way to programmatically generate a new model (see 7.6).

Parsing a model requires a `SystemModelParser` and an input source (i.e. an `InputStream`, a `File` or a `Path`). Figure 6.1 shows an example of how to do this.

```
// [...]
  public static final SystemModel parseModel(String filename)
        throws IOException, SystemModelSerializationException {
    SystemModelParser parser =
            SystemModelParsers.newDefaultModelParser();
    SystemModel model;

    if (filename.equals("-")) {
        model = parser.parse(System.in, "<stdin>");
    } else {
        model = parser.parse(new File(filename));
    }
    return model;
  }
// [...]
```

Figure 6.1: Small example showing how to parse a model

```
  protected final static InputStream asStream(String... s) {
    return new ByteArrayInputStream(
      Joiner.on("\n").join(s).getBytes(StandardCharsets.UTF_8));
  }
```

Figure 6.2: Small example transforming an array of `Strings` to an `InputStream`

If the model specification is given in a `String` or an array of `Strings`, it is still possible to parse the specification by turning the (array of) `String`(s) into an `InputStream`. This technique was readily deployed for testing purposes (see figure 6.2).

6.1.2 Running simulations

Running a simulation on a model requires a bit more code than simply parsing a model. The main source of complexity in this task is setting up the actors.

Figure 6.3 contains an example of setting up and running a simulator with a single agent in it. The particular example assumes that it is possible to describe the desired outcome via an invariant (implemented as a `Predicate`).

It may be easier to describe the "end condition" and then turn that into an invariant. Usually, this is as simple as negating the end condition. The template in figure 6.4 might be useful for this purpose.

```
  SystemModel origModel = ...;
  SystemModelSimulator simulator =
    SystemModelSimulators.newSystemModelSimulator(origModel);
  SimulatedSystemModel simModel = simulator.getSimulatedModel();
  ActorAI ai = ...;
  Predicate<SimulatedSystemModel> invariant = ...;
  int limit = 100;
  /* Create a simulated actor for "john" (in the origModel) and
   * let him start at the node "outside".
   */
  simulator.createSimulatedActor("john", "outside").attachAI(ai);
  simulator.startSimulation();

  /* Let the AI perform a fixed number of steps */
  if (SimulatorUtil.runWhile(simulator, invariant, limit)) {
       /* Invariant held for "limit" calls to stepTime() */
// [...]
  } else {
       /* Invariant violated before "limit" */
       SystemModelTrace trace = simulator.viewCurrentTrace();
       SystemModelLog log = simulator.viewCurrentLog();
       // Review the trace or/and the log
// [...]
  }
```

Figure 6.3: Small example for setting up a simulator with one actor. The "...;"-parts are to be filled out by the consumer of the example.

```
/* Returns true when the actor has completed his objective
 * or a policy violation has occurred etc.
 */
Predicate<SimulatedSystemModel> endCondition = ...;
/* ... negating endCondition check turns it into an invariant. */
Predicate<SimulatedSystemModel> invariant =
    Predicates.not(endCondition);
```

Figure 6.4: Snippet showing how to create an invariant from an "end condition".

It is also possible to control the simulation more finely grained by using the simulator's `stepEvent` and `stepTime` methods.

6.2 Implementing a fixpoint analysis

All fixpoint analyses in the framework should implement the `FixPointAnalysis` interface. It is basically implemented as a "visitor" (as in "visitor-pattern"). The basic setup is as the following.

The analysis has to provide the fixpoint analyser with two things. First, a collection of start nodes and secondly, a direction (either "forwards" or "backwards"). From there on the analyser will make analysis will visit related nodes one or more times. Finally, the analyser will inform the analysis when the fixpoint has been reached, so it can do any final processing.

However, most analysis implementations are probably going to do one or more of the following:

- Associate each (visited) node with some data.
- Special-case the first visit to a given node (especially true for start nodes).
- Keep track of "reachable" nodes (the code refers to these as "successfully visited nodes").

Therefore, the framework also provides an `AbstractFixPointAnalysis` class, which provides these features.

6.2.1 A simple reachability analysis

I have implemented a "simple" reachability analysis, which is based on the `AbstractFixPointAnalysis`. I write "simple" because it turns out that "correct" handling of reachability is not actually really simple.

The full code of the class is far too long to be embedded here, but is available in the appendix A.1. As I decided to use it as an example, it has been extensively documented with corner cases and rationales for pretty much every case I could think of. There are even some corner cases it handles poorly performance-wise, which I have left as-is for now.

```
@Override
protected NodeAnalysisResult reanalyseNode(SystemGraphNode node,
    Optional<ReachabilityData> optData) {
  /* We have processed this node before (successfully).
   * One of our predecessors must have learned something new. */
  ReachabilityData data = optData.get();
  NodeAnalysisResult res = updateDataFromPredecessors(node, data);
  /* Even if our predecessors learned something new, we are not
   * actually guaranteed thats its knowledge will reach this node.
   * Example:
   *
   *   predecessor --- (impassible edge) --> current node
   *
   * The fixpoint analyser does not know our criteria for
   * whether a node is reachable or not, so it just submits the
   * node for us to process and let us deal with it.
   */
  if (res == NodeAnalysisResult.CHANGED) {
    /* If we learned something new (likely) try to see if we
     * can obtain more data on this node.
     *
     * XXX: Technically, we only need to do this if we obtained
     * a new credential from our predecessors (since documents
     * do not "unlock" access to nodes).  This optimisation is
     * left as an exercise to the reader of this comment.
     */
    updateDataFromSuccessors(node, data);
  }
  return res;
}
```

Figure 6.5: The implementation of reanalyseNode

The implementation relies on `AbstractFixPointAnalysis` to keep track of which nodes are reachable. As a side effect, it also keeps track of the nodes for which the analysis have data. This means that `reanalyseNode` is basically reduced to 5 lines of code with a single if-statement (see figure 6.5).

The analysis works by processing each node in basically two steps. First it looks "backwards" from the current node to see what data is available from the direct predecessors. This is done by the method `updateDataFromPredecessors`, which is included in figure 6.6. This step also doubles as reachability test for new nodes.

If this step concludes that new data is available, then the analysis will see what possessibles can be read from successor nodes of the current node. This is done in a method called `updateDataFromSuccessors`.

```java
private final NodeAnalysisResult updateDataFromPredecessors(
    SystemGraphNode node, ReachabilityData data, boolean newNode){
  NodeAnalysisResult res = NodeAnalysisResult.NOT_CHANGED;
  Predicate<SystemGraphNode> reachablePred =
      getSuccessfullyVisitedNodesPredicate();
  Iterable<? extends SystemGraphNode> predecessors =
      Iterables.filter(node.getPredecessorNodes(), reachablePred);

  for (SystemGraphNode predecessor : predecessors) {
    SystemGraphEdge e = predecessor.getOutgoingEdge(node);
    ReachabilityData srcData = getNodeData(predecessor).get();

    if (!canMoveToTarget(e, srcData.credentials)) {
      /* Not possible to move across this edge */
      continue;
    }
    if (newNode) {
      /* This case may seen a bit weird, but it is a special case
       * for when there are no initial credentials.  In a graph
       * like
       *
       *   outside -> hall_way -> ...
       *
       * Then if outside is the start position, then it will be
       * unconditionally be marked as changed (for being the start
       * position).
       *
       * But hall_way will not be marked as changed if there are
       * no reachable credentials/documents at "outside".  The
       * problem is, if there are no credential requirements for
       * going from outside to hall_way, then we still want to see
       * what we can reach from hall_way.
       */
      res = NodeAnalysisResult.CHANGED;
    }
    if (data.credentials.addAll(srcData.credentials)
        || data.documents.addAll(srcData.documents)) {
      res = NodeAnalysisResult.CHANGED;
    }
  }
  return res;
}
```

Figure 6.6: The code for merging new data from predecessors.

That makes up the basics behind the simple reachability analysis provided by the framework.

The not so simple part. Unfortunately, the second step is where the analysis stops being simple. The basic problem is that if a new credential becomes reachable, it may make another credential reachable. So to do this correctly, one has to do a fixpoint computation to find all newly available credentials.

It is not difficult to do, but it is definitely a bit more complex than I wanted it to be. The relevant code parts can be found in the methods `updateDataFromSuccessors` and `findNewCredentials` (see the full source in appendix A.1). These methods also include inline comments describing the cases where the fixpoint computation is needed.

6.3 Implementing an AI or Jung agent

The simulator interfaces with all agents via the `ActorAI` interface. The implementation "simply" has to provide a suitable implementation for `pollAction` and `actionFailed`.

`pollAction` is invoked by the simulator once every time step and the AI have to return the action (i.e. an `ActorAction`) it wishes to perform in this time step. All actions are created via the static factory class `ActorActions`.

If the action fails, `actionFailed` is invoked to inform the agent for the failure. Regardless of whether the action failed or not, `pollAction` will be called next time the agent can perform a move.

The agent can also install a `SimulationUpdateEventListener` in the simulator, which will provide it access to all events that happen in the simulator as they are played out. For added realism, the agent might filter out the events that it "could not have reasonably known about".

Agents that are willing to cooperate with others can also implement the `CooperativeActorAI` interface. This gives other agents a simple standardised interface for sharing possessibles between each other.

6.3.1 A document stealing thief

I had the pleasure of working with Emil Gurevitch, who implemented the first non-trivial agent in this framework. The implementation features an (almost) deterministic and memory-less prototype agent capable of stealing documents and credentials.

The agent generally have 3 major active states, which are:

1. Find the document
2. Find a credential
3. Move to a pre-defined "extraction" point

Once it is at the extraction point with the document or it concludes it cannot accomplish its mission, it goes to a passive "do nothing" state[1]. But as mentioned, the agent is memory-less, so it always have to recompute its current intention from its current state. Accordingly, it never remembers paths. These are recomputed between any two calls to `pollAction`. Admittedly, here the framework lacks a few utilities (see subsection 7.8).

On the plus side, this means that the agent will automatically get "unstuck" if another agent provides it with an opportunity to continue its mission.

As mentioned, it is mostly deterministic, which makes it easier to debug. The only source of non-determinism (that I am aware of) is that it relies on the determinism of path finders. It uses the built-in "breadth-first" path finder, which is deterministic only if there is exactly one "shortest path". This is usually pretty easy to accommodate during tests and most debugging sessions.

The agent is not smart, but does not try to be it either. The agent will basically start in state 1 ("find document") and then go to state 3 ("go to exit") once it has the target document. State 2 ("find a credential") is only used if it cannot complete its current state.

The lack of "smartness" becomes strikingly obvious once you realise that "find a credential" is literally "find *any* credential" rather than "find this credential that I need" (see figure 6.7). Admittedly, this keeps the agent fairly simple as it does not have to compute which credentials are needed. This can in theory be

[1]Technically, internally these are 2 distinct states, but the difference between the two is not interesting to understand the agent

```
private ActorAction obtainACredential(final SimulatedActor actor){

  /* Sort it in order to take the same credential each time. */
  List<? extends Possessible> allCreds = new ArrayList<>(
      model.getAllKnownCredentials());
  if (allCreds.isEmpty()) {
    throw new IllegalStateException("no known credentials found");
  }
  Collections.sort(allCreds, new StringCompare());
  for (Possessible cred : allCreds) {
    try {
      return obtainPossessible(actor, cred);
    } catch (IllegalStateException exc) {
      continue;
    }
  }
  throw new IllegalStateException("could not find an allowed"
    + " path to any of the " + allCreds.size()
    + " credential(s) found.");
}
```

Figure 6.7: `obtainACredential` from the Thief agent

a rather daunting task and I suspect the framework could use an improvement of some of its utilities to make this easier.

With this simple approach, the code determining if the agent can obtain a given `Possessible` is basically reduced to finding a path to the `Possessible`. This implementation is shown in figure 6.8. If that fails, the agent simply ignores the `Possessible` for the moment and tries to obtain a (different) credential.

6.3.1.1 Result of using the agent on a simple model

Using the `SimulatorExample` program from the "examples" component, I ran the thief AI on a very simple model. The specification of the model is available in figure 6.9. The output of the program, including the trace and the log file, is available in figure 6.11.

A visual representation of the model is available in figure 6.10. The visual representation was generate with graphviz component by Emil Gurevitch with a few minor manual changes.

On that model, the thief will manage to enter the building, steal the "doc" document and escape outside in 5 steps. Thanks to the trace, we can see every step the thief takes. Furthermore, we see one log entry when the thief enters

```
private List<SystemGraphEdge> findPathToPossessible(
      final SimulatedActor actor, Possessible possessible,
      Set<SystemGraphNode> possessibleLocations) {
  SystemGraphNode actPos = actor.getPosition();
  Multiset<SystemCredential> actCred = actor.getCredentials();

  /* Find neighbor nodes to each possessible node, ignoring nodes
   * where it is not possible to perform the requested actions. */
  Set<SystemGraphEdge> targetEdges = new HashSet<>();
  for (SystemGraphNode posLoc : possessibleLocations) {
    Iterables.addAll(targetEdges, Iterables.filter(
        posLoc.getIncomingEdges(),
        PathFinderPredicates.hasSufficientCredentials(
            possessibleActions, actCred.elementSet())));
  }
  Set<SystemGraphNode> targetLocations = new HashSet<>();
  for (SystemGraphEdge edge : targetEdges) {
    targetLocations.add(edge.getSourceNode());
  }
  if (targetLocations.isEmpty()) {
    throw new IllegalStateException("not currently possible"
      + " to get access to this possessible: empty set of"
      + " reachable neighbor candidates");
  }

  /* Find path to a neighbor node. */
  GraphPath<SystemGraphNode, SystemGraphEdge> pathToTarget =
      finder.findPathEdgesToAnyDest(
        actPos,
        targetLocations,
        PathFinderPredicates.hasSufficientCredentials(
            ActorActionType.MOVE,
            actCred.elementSet())).orNull();
  if (pathToTarget == null) { // no path could be found.
    throw new IllegalStateException("search yielded no path to"
        + " possessible: " + possessible);
  }
  return pathToTarget.getTraversedEdges();
}
```

Figure 6.8: `findPathToPossessible` from the Thief agent

```
locations {
  container: [safe],
  physical: [outside, hall, office],
}
connections {
  outside -> hall {*: m};
  hall -> office {*: mlog};
  office -> safe {*: r, o};
  office -> hall {*: m};
  hall -> outside {*: m};
}
actors {
  thief @ outside;
}
documents {
  doc @ safe;
}
```

Figure 6.9: Small example model used for testing the thief AI

the office.

The log entry tells us that “an unknown actor” moved from the “hall” node to the “office” node. This action did not involve any credentials or documents being moved or shared (“[no-possessible]”).

What may come as a surprise, is that the thief attempts to use a credential to enter the office. The model does not require any credentials for this action. This is an artefact of the thief using every credential it possesses in every action it takes. But it is also an artefact of the framework logging all credentials thrown at it, even where it might not make sense to log these credentials.

The simple logging format from the program does not actually say which credential was used. However, in this model, there is only one credential - the identity credential of the thief. In other words, the thief effectively exposed itself directly in the log.

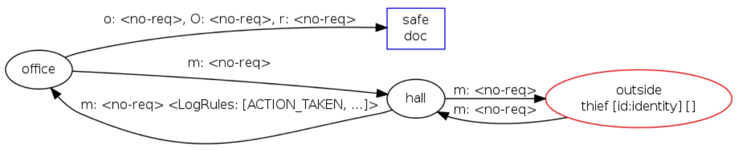
o: <no-req>, O: <no-req>, r: <no-req>
safe
doc
office
m: <no-req>
hall
m: <no-req>
m: <no-req>
outside
thief [id:identity] [ ]
m: <no-req> <LogRules: [ACTION_TAKEN, ...]>

Figure 6.10: Visual representation of the example model. Auto-generated from Emil Gurevitch's graphviz component with a few manual changes

```
thief obtained doc in 5 time steps
 [1] move from outside to hall
 [2] move from hall to office
 [3] read (copy) doc from safe
 [4] move from office to hall
 [5] move from hall to outside
Format of the log entries:
  [<time>] Entry: <actor> || <action-type> || <source-node>\
  || <target-node> || <possessible> || <credentials-used>
Log file contained:
 [2] Entry: [actor N/A] || MOVE || hall\
 || office || [no-possessible] || 1 credential used
```

Figure 6.11: Results for the thief AI in a small model. Long lines have been wrapped. Where this occurs, a "\" will occur where the line was split.

```
// [... all locations ...]
connections {
   office -> safe {
         U.safe_key: i (0%, 1);
         *: i (10%, 3);
   };
   // [... more connections ... ]
}
// [... rest of the model specification ...]
```

Figure 6.12: Example of annotations in the model language

6.4 Risk and time analysis

The extended version of the model language supports annotations on the access control specification. These annotations can be used to describe "detection risk" and "time-cost" of actions. A small example of the syntax can be seen in figure 6.12:

This particular example declares that there are two ways to access the safe from the office. One is if you have "U.safe_key" - in this case, there is a detection risk of 0% and a time-cost of 1. Alternatively, you can access the contents of the safe without using a key, but then you have 10% detection risk and time-cost of 3.

These values are only informative, in the sense that nothing in the framework enforces these values. This is also why they are referred to as "annotations" in the actual framework. Integrating this better has been left as future work (see 7.2).

While they are only informative, they can still be used for any kind analysis like creating plans for "minimal risk" or "least time spent". As an example, figure 6.13 shows how to sum up the time-cost for a given path.

6.5 Usage conclusion

This section showed various examples of what this framework can be used for and how to do it. Ranging from simple things like parsing models to static fixpoint analysis and risk/time analysis.

```
GraphPath<SystemGraphNode, SystemGraphEdge> movePath = ...;
Iterable<? extends SystemAccessPolicy> movePolicies
    = PathUtilities.mapPathToMovePolicies(movePath);
Iterable<? extends SystemAccessPolicy> nonEmptyMovePolicies
    = Iterables.filter(movePolicies,
                  PathFinderPredicates.nonEmptyPolicyPredicate());
String k = SystemAccessPolicyComponent.ANNOTATION_KEY_ACTION_TIME;
int timeCost = movePath.size();  /* One for each edge */
for (SystemAccessPolicy policy : nonEmptyMovePolicies) {
    ImmutableSet<SystemAccessPolicyComponent> comps
         = policy.getPolicyComponents();
    SystemAccessPolicyComponent comp
         = [... choose one from comps ...];
    SystemComponentAnnotation anno = comp.getAnnotation();
    /* Get the cost of this particular edge which may be more
     * expensive than a regular edge.  We substract one to
     * avoid over-counting (we already "paid" 1 for all edges
     * before the loop).
     */
    timeCost += anno.getValueAsInt(k, 1) - 1;
}
```

Figure 6.13: Example showing how to use annotations to get the time-cost for a path

CHAPTER 7

Future work

This section covers some of the things that are not (fully) implemented in this framework. It also mentions a couple of things that could be useful to add in the future.

7.1 The "exec" action

I deliberately left out full support for the "execute" action. The most important reason for leaving it out was that its semantics were not clear to me. At the moment, a large part of the framework knows that the execute action exists, but none of the code knows what to do when it sees such an action.

To my knowledge, support for the "execute" action requires a minimum of three changes:

- An `ActorAction` to support the "execute" action needs to be implemented.
- A static factory method in `ActorActions` for instantiating the action.
- Implementation of its semantics in the simulator.

From there, agents will be able to use the "execute" action as they please. There may be other parts of the code needing updates, like `SimulatorExample`. However, these should be easy to find as they will generally start to throw `UnsupportedOperationException` or `AssertionError` when faced with an "execute" action.

7.2 Integration of "detection-risk" and "time-cost"

At the moment, this extra information is just attached on top of the models. Most of the existing implementation simply ignores this information. This (among things) allows "time-skew" in the simulator, where actors always complete their actions in one time-step regardless of what the "time-cost" says.

For proper integration of these values, their semantics need to be defined. Like how is "detection-risk" aggregated over a sequence of actions? It is also quite possible that it is better to leave these under specified at the moment. This would at least allow people to apply their own interpretation of them and see where it leads them.

7.3 Solve the "copy" problem

Currently every actor is trivially able to copy any document or credential via a non-destructive read, write or share action. The latter does require a collaborating agent, but that agent need not realise that it is used as a "moving copier".

For digital media and passwords, this makes sense (within a reasonable limit). Coping a computer file to an USB stick or writing down a password on a piece of paper is usually fairly trivial. However, when dealing with a physical media this copying becomes somewhat awkward.

For a physical document, one could in theory write a manual copy of it (although it will take some time and possibly quite a bit of paper). Alternatively, in this day and age, one could assume they had a small camera that could take pictures of each piece of paper. I suspect the latter is a reasonable interpretation of how an agent can copy a physical document. Admittedly, it does not explain how they are later able to put that copy on a desk in physical form without passing by a printer first…

But the true problem is physical credentials. Your average "John Doe" does not carry a device to make copies of a physical key or a keycard in his pocket. Or a device to copy an ID badge but replace the picture on it. But in the current model, they can.

A "quick fix" to this, might be to simply disallow any copy operation on physical credentials. It would not change the current semantics for documents, but they are probably "mostly fine" (or at least, they seem more reasonable).

7.4 Revise the "per possessible" access policies

Currently the "per possessible" access policies have the same capabilities as their "per edge" counterparts. Here I diverge from Probst and Hansen, who had a separate set of "DataActions"[6, p. 240 (fig. 5)]. The major deficiency is that the decrypt actions are not supported. But also many of the "per edge" access rules do not make a lot of sense in an access rule for a possessible. I think the parser would do well to reject e.g. a "move" capability in an access rule for a possessible.

This difference started as an artefact from the implementation and I simply never allocating the time to deal with it. By now, changing it will probably break some parts of the current API.

If I had to solve this myself, I would probably change the `SystemGraphEdge` API. Namely, replace (or deprecate):

```
public Iterable<? extends SystemAccessPolicy> getRequiredPermission(
                ActorActionType t, Possessible data)
                throws IllegalArgumentException;

public boolean  hasSufficientCredentials(ActorActionType t,
                Possessible data, Collection<SystemCredential> cred)
                throws IllegalArgumentException;
```

with something like:

```
public Iterable<? extends SystemAccessPolicy> getRequiredPermission(
                ActorActionType t, DataActionType dt,
                Possessible data)
                throws IllegalArgumentException;
```

```
public boolean hasSufficientCredentials(ActorActionType t,
                Possessible data, DataActionType dt,
                Collection<SystemCredential> cred)
                throws IllegalArgumentException;


// either
/* Map an ActorActionType to a set of possible DataActionType
   for that ActorActionType */
public ImmutableSetMultimap<ActorActionType, DataActionType>
                getPossibleActionsForPossisble(Possible data)
                throws IllegalArgumentException;

// or
/* Return a (possibly empty) set of possible DataActions */
public ImmutableSet<DataActionType> getPossibleDataActions(
                ActorActionType t, Possible data)
                throws IllegalArgumentException;
```

And then introduce a `DataActionType` enum to enumerate all the possible data actions. For decryption, it may need some changes to the `SystemDocument` interface to allow actors to obtain an encrypted document.

This would definitely also have consequences for the parser and the `SystemModelBuilder`. Fortunately, those changes are completely “internal” to the implementation and should not affect the consumers of the framework.

At the same time, it might be prudent to add support for actors starting with documents. This is currently not supported either.

7.5 Undo-capable simulator

I originally envisioned the simulator being able to go backwards as well. That is, allow undo actions. This would require all AIs to support the notion of being “rewinded”. I would not require them to do exactly the same actions as they did before - that would “just” be a “replay” of the trace.

Rather, I would want to allow them to try something else when steps were undone. Admittedly, this probably makes more sense for AIs that are (partially) guided by humans (or maybe those that rely on machine learning). Alternatively, it could allow one to try the same scenario again but change the outcome of “conflicts” differently.

However, I never got around to implement the undo-capable simulator. Furthermore, the interface to the AI might need to be revised for this to fully work.

7.6 Mutable and serialisable models

Currently, the only (supported) way of obtaining a model is by parsing it from a specification. This is trivial to do - in fact it is so easy that most of the included tests cases involves parsing a new model. But there is one major use-case not supported by this. Creating of new models from scratch.

There is absolutely no support for creating a "model editor", i.e. a program to help users build models. Basically, two things are required (besides the program itself). One is an implementation of a "mutable" model and the other is a serialiser for said model. Possibly a parser/deserialiser and a less rigid model format for these incomplete models would be useful as well.

Technically speaking, the "unfinished" model need not implement the `SystemModel` interface. However, it would naturally allow the editor trivial access to all the existing tools if the unfinished model reused the same interface.

7.7 Visualisation of models

A very useful feature, which is sadly not present, is to visualise the models. It would make a lot of things easier to review (for humans) and is quite possibly required for writing a graphical interface for tools working on these models.

Here, it might be possible to merge back some of the utilities Emil Gurevitch created. API-wise they do leave a bit to be wanted, but they do look promising at first glance.

7.8 Better support for AIs

Currently AIs have a paper thin interface between them and the simulator. While it works fine, it means that most agents will probably be implemented inefficiently or re-implement the same utilities (e.g. for maintaining plans).

As an example, when going from node A to node B the path finder will provide a list of edges to traverse. The agent then has to transform this list of edges into actions. But when it has the list of actions, it also has to maintain the list over repeated calls to `pollAction`. Since the latter inherently involves fiddling with state, some of these "reimplementations" are bound to have bugs.

If this plan-maintenance was supported by the framework, all agents could reuse it. Furthermore, the bugs (if any) would only need to be fixed once.

7.9 Make large models maintainable

As mentioned in subsection 4.2.2, I believe the model language is insufficient for describing and maintaining very large models. It is a short-coming for which I have not been able to find a full solution.

It could be worth it to split such models across multiple files. It might also be worth it to look into creating a more specialised language for the infrastructure part.

For prototyping, the current parser should be sufficient as long as the revised language can be rewritten into the current model format. That said, since the current parser constructs a full abstract syntax tree, it might consume very large amounts of memory for very large models.

7.10 Future work conclusion

This section has elaborated on some of the major areas, where the framework is currently feature incomplete. It also mentioned some places where there are unresolved issues with the model semantics. This includes problems such as missing support for the "exec" action, the missing "data actions" for access rules on possessibles and the "copy" problem.

Chapter 8

Conclusion

I believe that system models have yet to reach their full potential yet. My analysis of the current situation led me to conclude that system models needed an extensible framework to facilitate new work in this area.

Leveraging on the experiences of Joshua Bloch, I setup some guidelines for how I wanted to develop the framework. I also noticed some aspects of the current model specification language that I found inadequate.

From this I implemented a framework to work on and with system models. The framework features a parser for a revised model specification language that fixes some of the inadequacies in its predecessor. In particular, a lot of the boilerplate syntax was reduced and actor roles were added to make large scale models easier to maintain. The new language also features annotations to declare e.g. "time-cost" of actions.

Using the framework, I wrote a static reachability analysis. Emil Gurevitch wrote a simple document stealing AI. Furthermore, I wrote several basic examples for solving common use cases as well as a demonstration of how to use the "time-cost" annotation value.

While the implementation provides a foundation for working with system models, it still leaves some features to be implemented, like the "execute" action and visualisation of the models. There are also a couple of inadequacies in the language like missing data access rules and the “copy” problem.

Appendix A

Appendix

A.1 SimpleReachabilityAnalysis.java

This is full code for the "SimpleReachabilityAnalysis" class with the exception of the import-section being filtered out.

```
package dk.dtu.imm.sysmodel.analysis;
// [...]
/**
 * <p>This is a simple reachability analysis.  Given a set of
 * credentials and a starting position, this analysis will deduce
 * what positions are reachable from that starting position.
 * Furthermore, at each reachable position, it will determine
 * which credentials and documents are reachable there.</p>
 *
 * XXX: Currently this analysis only considers move actions and read
 * actions.
 */
@NonNullByDefault
public final class SimpleReachabilityAnalysis extends
AbstractFixPointAnalysis<SystemGraphNode, SystemGraphEdge,
/* Marker to keep indentation*/ SystemModel, ReachabilityData> {

  /**
   * Value for denoting that no credentials to be obtained.  This
   * is a value (namely, 0) and not a bit flag.
   */
```

```
private static final int NO_NEW_CREDENTIALS_ON_TARGET = 0x00;

/**
 * Bit flag that denoting that the target node had new
 * credentials, but they were not reachable.
 */
private static final int FLAG_HAD_UNREACHABLE_CREDENTIALS = 0x02;

/**
 * Bit flag that denoting that the target contained new
 * credentials and at least one of them were reached.
 */
private static final int FLAG_OBTAINED_NEW_CREDENTIALS = 0x01;

/**
 * The minimum credentials available on the starting positions
 */
private final ImmutableSet<SystemCredential> initialCredentials;

/**
 * Create a new reachability analysis
 *
 * @param initialCredentials The credentials available at
 *  the starting position.
 * @param startPosition The starting position.
 */
public SimpleReachabilityAnalysis(
      Iterable<SystemCredential> initialCredentials,
      final SystemGraphNode startPosition) {
  super(ImmutableSet.of(startPosition),
        AnalysisDirection.FORWARDS);
  this.initialCredentials =
        ImmutableSet.copyOf(initialCredentials);
}

@Override
protected NodeAnalysisResult analyseNewNode(SystemGraphNode node,
      Optional<ReachabilityData> inData) {
  /* We are visiting a node for the first time */
  ReachabilityData data;
  NodeAnalysisResult res;
  assert !inData.isPresent();

  if (startNodes.contains(node)) {
      /* It is a start position */
      data = new ReachabilityData(initialCredentials);
      /* If start node has other start nodes as neighbours,
       * updateInputData might learn something new.
       * (Unlikely, but we need it for correctness)
       */
      if (startNodes.size() != 1) {
          /* Technically, this analysis only have one start node,
           * so this case is impossible. But this is an example
           * and someone will probably copy-waste it as-is, add
           * multiple start nodes and forget about this case.
```

```
                 *
                 * To "that copy-wasting someone":  You are welcome. :)
                 */
                updateDataFromPredecessors(node, data, true);
            }
            /* We always consider the start node as having changed
             * when we visit it the first time.
             */
            res = NodeAnalysisResult.CHANGED;
        } else {
            /* This is a node we have not seen before and it is not
             * a start position */
            data = new ReachabilityData();
            /* We may not be able to visit this node yet.  In that case,
             * updateDataFromPredecessors will return NOT_CHANGED and
             * we will just return that.  Our parent class will then
             * "unmark" the node.  See the documentation of the parent
             * class for more information.
             */
            res = updateDataFromPredecessors(node, data, true);
        }

        if (res == NodeAnalysisResult.CHANGED) {
            /* We are able to visit this node, so insert the data */
            setNodeData(node, data);
            /* With the initial credentials obtained via our
             * predecessors, see what we can obtain from this
             * node.
             */
            updateDataFromSuccessors(node, data);
        }
        return res;
    }


    @Override
    protected NodeAnalysisResult reanalyseNode(SystemGraphNode node,
            Optional<ReachabilityData> optData) {
        /* We have processed this node before (successfully).  One
         * of our predecessors must have learned something new. */
        ReachabilityData data = optData.get();
        NodeAnalysisResult res = updateDataFromPredecessors(node,
                data, false);
        /* Even if our predecessors learned something new, we are
         * not actually guaranteed thats its knowledge will reach
         * this node.  Example:
         *
         *    predecessor --- (impassible edge) --> current node
         *
         * The fixpoint analyser does not know our criteria for
         * whether a node is reachable or not, so it just submits
         * the node for us to process and let us deal with it.
         */
        if (res == NodeAnalysisResult.CHANGED) {
            /* If we learned something new (likely) try to see if we
         * can obtain more data on this node.
         *
         * XXX: Technically, we only need to do this if we obtained
         * a new credential from our predecessors (since documents
         * do not "unlock" access to nodes).  This optimisation is
         * left as an exercise to the reader of this comment.
         */
        updateDataFromSuccessors(node, data);
    }
    return res;
}

/**
 * Check predecessors of a node for new credentials or documents
 *
 * <p>When visiting a node, its predecessors will determine which
 * credentials and documents are available at this node.  The sole
 * exception to this rule are starting positions (which always
 * have the initial credentials available regardless of its
 * predecessors).
 * </p>
 * <p>Since this is a fixpoint analysis, new knowledge is
 * propagated via nodes over multiple iterations.  This method
 * takes care of reviewing the predecessors for initial or new
 * knowledge and inserting it into the data element for this
 * node.
 * </p>
 *
 * @param node The node being visited.
 * @param data The analysis data for this node.  This may be
 * updated by this call.
 * @param newNode True if this is a new node.  In this case, it
 * will be marked as {@link NodeAnalysisResult#CHANGED}, if it
 * is reachable from any predecessor (even if it provides no new
 * data).
 * @return {@link NodeAnalysisResult#CHANGED} if one (or more) of
 * predecessors of this node provided a new credential or document
 * to become available on this node.  Returns
 * {@link NodeAnalysisResult#NOT_CHANGED} otherwise.
 */
private final NodeAnalysisResult updateDataFromPredecessors(
        SystemGraphNode node, ReachabilityData data,
        boolean newNode) {
    NodeAnalysisResult res = NodeAnalysisResult.NOT_CHANGED;
    Predicate<SystemGraphNode> reachablePred =
            getSuccessfullyVisitedNodesPredicate();
    Iterable<? extends SystemGraphNode> predecessors =
            Iterables.filter(node.getPredecessorNodes(),
                    reachablePred);

    for (SystemGraphNode predecessor : predecessors) {
        SystemGraphEdge e = predecessor.getOutgoingEdge(node);
        ReachabilityData srcData = getNodeData(predecessor).get();

        if (!canMoveToTarget(e, srcData.credentials)) {
                /* Not possible to move across this edge */
                continue;
            }
            if (newNode) {
                /* This case may seen a bit weird, but it is a special
                 * case for when there are no initial credentials.
                 * In a graph like
                 *
                 *   outside -> hall_way -> ...
                 *
                 * Then if outside is the start position, then it will
                 * be unconditionally be marked as changed (for being
                 * the start position).
                 *
                 * But hall_way will not be marked as changed if there
                 * are no reachable credentials/documents at "outside".
                 * The problem is, if there are no credential
                 * requirements for going from outside to hall_way,
                 * then we still want to see what we can reach from
                 * hall_way.
                 */
                res = NodeAnalysisResult.CHANGED;
            }
            if (data.credentials.addAll(srcData.credentials)
                    || data.documents.addAll(srcData.documents)) {
                res = NodeAnalysisResult.CHANGED;
            }
        }
        return res;
    }

    /**
     * Check if new data can be obtained from successor nodes
     *
     * <p>When it has been determined that the current node has
     * obtained a new credential, this method will check for new
     * reachable {@link Possessible}s in successor nodes.  It does
     * not return a {@link NodeAnalysisResult}, because when this
     * is called it has already been determined that the current
     * node has changed.
     * </p>
     *
     * @param node The node being visited.
     * @param data The analysis data for this node.  This may be
     * updated by this call.
     */
    private static final void updateDataFromSuccessors(
            SystemGraphNode node, ReachabilityData data) {
        Iterable<? extends SystemGraphEdge> edgesToCheck =
                node.getOutgoingEdges();
        boolean recheck = false;

        do {
            /* Do a little fixpoint analysis to obtain all credentials
             * from successors.  This little fixpoint runs in
```

```
         * O(|E| * R), where |E| is the number of outgoing edges
         * and R is the number of iterations needed to reach a
         * fix point.  Worst case, R will be the number of
         * credentials reachable from this node.
         *
         * However, the assumption is that credentials are not lying
         * around in big piles (with incremental access controls).
         * So in practise, R will be fairly low.  Worst case
         * scenarios looks something like this:
         *
         *   safe1 contains the key for safe2
         *   safe2 contains the key for safe3
         *   ...
         *   safe(N-1) contains the key for safeN
         *
         *   All safes are reachable from the same node and we
         *   always process the safes in the worst possible
         *   order (e.g. starting from safeN to safe1).
         *
         *   key for safe1 is readily available from the same node.
         *
         * It seemed like a ridiculously case, so I have not
         * bothered optimising the algorithm for it.
         *
         * XXX: Optimising this code so it less vulnerable to the
         * above case is an exercise left to the reader.
         */
        List<SystemGraphEdge> nextRecheckList = null;

        for (SystemGraphEdge e : edgesToCheck) {
            int result = findNewCredentials(e, data.credentials);
            if ((result & FLAG_HAD_UNREACHABLE_CREDENTIALS) != 0) {
                /* There were credentials left on the target node.
                 * Remember the edge so we can re-process it again
                 * if we find a new credential later.
                 *
                 * Note we do this even if
                 * FLAG_OBTAINED_NEW_CREDENTIALS is set.  This is
                 * because findNewCredentials does not compute a
                 * fixpoint (and we are already set up for doing
                 * it anyway, so we might as well do it here)
                 */
                if (nextRecheckList == null) {
                    nextRecheckList = new ArrayList<>();
                }
                nextRecheckList.add(e);
            }

            if ((result & FLAG_OBTAINED_NEW_CREDENTIALS) != 0) {
                /* We obtained a new credential, schedule a re-check
                 * if we had any edges with unreachable credentials.
                 *
                 * NB: We deliberately do this after the check for
                 * unreachable credentials.  (See the "unreachable
                 * credentials" case above).
                 */
                recheck = nextRecheckList != null;
            }
        }

        /* remember to update edgesToCheck */
        edgesToCheck = nextRecheckList;
    } while (recheck && edgesToCheck != null);

    /* Now that we obtained all the new credentials, look at what
     * documents we can reach. We do not need all the heavy
     * lifting of the above, since documents do not grant access
     * to anything.
     */
    for (SystemGraphEdge e : node.getOutgoingEdges()) {
        SystemGraphNode target = e.getTargetNode();
        for (SystemDocument d : target.getDocuments()) {
            if (data.documents.contains(d)) {
                continue;
            }
            if (canRead(e, d, data.credentials)) {
                data.documents.add(d);
            }
        }
    }
}

/**
 * Get the available credentials at a given node
 *
 * <p>This method returns the resulting </p>
 *
 * @param node A given node
 * @return A {@link Set} of the {@link SystemCredential} that can
 * be obtained at the given node.  The set is unmodifiable.  It
 * is may or may not be a view.
 */
public Set<SystemCredential> getObtainableCredentialsAtNode(
        SystemGraphNode node) {
    Optional<ReachabilityData> data;
    Preconditions.checkState(hasStarted(),
            "The analysis has not started yet");
    data = getNodeData(node);
    if (!data.isPresent()) {
        return ImmutableSet.of();
    }
    return Collections.unmodifiableSet(data.get().credentials);
}

/**
 * Get the available documents at a given node
 *
 * @param node A given node
 * @return A {@link Set} of the {@link SystemDocument} that can
 * be obtained at the given node. The set is unmodifiable.  It
 * is may or may not be a view.
 */
public Set<SystemDocument> getObtainableDocumentsAtNode(
      SystemGraphNode node) {
   Optional<ReachabilityData> data;
   Preconditions.checkState(hasStarted(),
           "The analysis has not started yet");

   data = getNodeData(node);
   if (!data.isPresent()) {
       return ImmutableSet.of();
   }
   return Collections.unmodifiableSet(data.get().documents);
}

/**
 * Test if a possessible can be read from an edge.
 *
 * <p>Helper method to keep length of lines down and conditions
 * simple.</p>
 *
 * @param e An edge
 * @param p A possessible on the target node of the edge
 * @param creds A set of credentials
 * @return true if p can be read over e with the given credentials
 */
private static final boolean canRead(SystemGraphEdge e,
      Possessible p, Set<SystemCredential> creds) {
   boolean copy = e.hasSufficientCredentials(
           ActorActionType.INPUT_COPY, p, creds);
   if (copy) {
       return true;
   }
   return e.hasSufficientCredentials(
           ActorActionType.INPUT_ORIGINAL, p, creds);

}

/**
 * Test if it is possible to move over a given edge
 *
 * <p>Helper method to keep length of lines down and conditions
 * simple.</p>
 *
 * @param e An edge
 * @param creds A set of credentials
 * @return true if it is possible to move over e with the given
 * credentials
 */
private static final boolean canMoveToTarget(SystemGraphEdge e,
      Set<SystemCredential> creds) {
   return e.isActionTypePossible(ActorActionType.MOVE)
          && e.hasSufficientCredentials(
                 ActorActionType.MOVE, creds);
}
```

```
/**
 * Check a target node for new credentials
 *
 * @param e The between the current node (source) and the target
 * node possibly containing new credentials.
 * @param creds A mutable set containing the current available
 * credentials.  It will be updated if new credentials become
 * available.
 * @return Either {@link #NO_NEW_CREDENTIALS_ON_TARGET} or a bit
 * mask of containing one or more of
 * {@link #FLAG_OBTAINED_NEW_CREDENTIALS} and
 * {@link #FLAG_HAD_UNREACHABLE_CREDENTIALS}.
 * <p><em>Note</em>: If both
 * {@link #FLAG_HAD_UNREACHABLE_CREDENTIALS} and
 * {@link #FLAG_OBTAINED_NEW_CREDENTIALS} are set in the return
 * value, it may be possible that a repeated call to this method
 * will obtain another credential.</p>
 */
private static final int findNewCredentials(SystemGraphEdge e,
        Set<SystemCredential> creds) {
  SystemGraphNode target = e.getTargetNode();
  int result = NO_NEW_CREDENTIALS_ON_TARGET;

  for (SystemCredential c : target.getCredentials()) {
      if (creds.contains(c)) {
          continue;
      }
      if (canRead(e, c, creds)) {
          /* This credential could give us access to another
           * credential accessible from this node (i.e. we ought
           * re-check all unreachable credentials on this node).
           * Though credentials are unlikely to lie around in
           * big piles, so we rely on our caller(s) to call us
           * again until they are happy.
           *
           * They pretty much have to any way, since someone
           * could write a model where like this:
           *
           *     safe1 contains all odd numbered keys between
           *           1 and N
           *     safe2 contains all even numbered keys between
           *           1 and N
           *
           *     To reach keyX, you need key(X-1).
           *     Agent starts with key0.
           *
           * In this case, you would (still) need N calls to
           * this method to obtain them all.  I believed this
           * case was pretty "unlikely" in  any "real" use, s
           * I have not optimised for it.  See also the comment
           * worst case runtime in updateDataFromSuccessors.
           */
          creds.add(c);
          result |= FLAG_OBTAINED_NEW_CREDENTIALS;
        } else {
            result |= FLAG_HAD_UNREACHABLE_CREDENTIALS;
        }
    }
    return result;
  }
}

/* Class holding the data obtained for a given node.  Its visibility
 * outside this class file is not needed (in particular, our
 * consumers need not know of it.
 *
 * Since it is an implementation detail, the class above gets to
 * access the members directly instead of using "accessors".
 */
final class ReachabilityData {
  /* reachable/obtainable documents at this node */
  final Set<SystemDocument> documents = new HashSet<>();
  /* reachable/obtainable credentials at this node */
  final Set<SystemCredential> credentials;

  ReachabilityData() {
    credentials = new HashSet<>();
  }

  ReachabilityData(Set<SystemCredential> initialCredentials) {
    credentials = new HashSet<>(initialCredentials);
  }
}
```

Bibliography

[1] Joshua Bloch. How to Design a Good API and Why it Matters. http://www.youtube.com/watch?v=aAb7hSCtvGw, 2007. [Online; accessed 2013-07-10].

[2] C. W. Probst and R. R. Hansen. Analysing access control specifications. In *Systematic Approaches to Digital Forensic Engineering, 2009. SADFE'09. Fourth International IEEE Workshop on*, pages 22–33. IEEE, 2009.

[3] Tollef Fog Heen. Sharing an SSH key, securely. http://err.no/personal/blog/tech/2013-03-22-09-45_sharing_an_ssh_key_securely, 2013. [Online; accessed 2013-07-29].

[4] Tobias Stig Lindø. Policy invalidation in system models. (IMM-PhD-2012-89), 2012.

[5] Oracle and/or its affiliates. Lambda Expressions (The Java(tm) Tutorials - Learning the Java Language - Classes and Objects). http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax, 2013. [Online; accessed 2013-07-29].

[6] Christian W. Probst and René Rydhof Hansen. An extensible analysable system model. *Information Security Technical Report*, 13(4):235–246, 2008. M3: 10.1016/j.istr.2008.10.012.

[7] Christian W. Probst, René Rydhof Hansen, and Flemming Nielson. Where can an insider attack. 2008.

[8] Simon Sinek. How great leaders inspire action. http://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action.html, 2009. [Online; accessed 2013-08-02].