Generating LTS for insider attack

Ibrahim Nemli

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Every organization or company relies on some data assets be it in one from or another, both digital assets and physical assets are present. One of the main challenges companies and organizations face is protecting these valuable assets. Many organizations and generally information security have focused on protecting assets from attacks perpetrated by people outside the boundaries of the targeted organization. This has been rather successfully by implementing firewalls, authorization mechanism, policies, encryption and etc. However, on the other side little research focus has been on protecting IT-infrastructures from insiders and dealing with insider attacks.

The objectives of the thesis is to present and extend a model used for detecting threats for insider attacks. The new extensions will incorporate new techniques for measuring probabilities for attacks that can occur in the system under consideration, by using methods from the discipline of static program analysis and model checking.

Preface

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

The thesis deals with the ExASyM framework (An **Ex**tensible **A**nalysable **Sy**stem **M**odel) that makes use of static analysis to analyse models of real-world systems and scenarios for insider threats. The output of the framework can be used to find weaknesses in real-world systems before an attack occurs, and also to guide the investigation of an attack after the attack has happened. This thesis will extends the ExASyM framework with new and more advanced capabilities and analysis techniques that involves probability, so one can ask questions about the likelihood for a certain attack will strike.

A software-tool will be implemented while others are used to serve the purpose of this project.

Lyngby, 02-February-2015

Ibrahim Nemli

Acknowledgements

During my education at DTU I had many interesting courses from the Language-Based Technology section which has highly motivated me to do a thesis that combined my knowledge learned from these courses.

I would like to thank my supervisor Christian W. Probst for proposing this very exciting project to me and for taking time to discuss my thesis.
Through out the period were this thesis was prepared I got very good advice and supervision in order to achieve the results presented in this report.

Contents

Chapter 1

Introduction

One of the major issues in information security amongst others is known as the "insider attack". It is an attack type that is perpetrated by malicious users who is already authorized access to a system. Thus, it makes it especially hard to detect these kind of attacks, as access to data assets are performed permissible. This leads to the fact that an insider has better abilities to compromise the security in an organization than an outsider may have due to his knowledge of high-value targets, weak points in the security and the fact that he is more trusted among his colleagues than outsiders may be. Consequently, an insider attack has the potential to cause significant, even catastrophic damage to the targeted organization. In addition, there may be less security against insiders because many organizations focus on protecting threats and attacks against external attackers. Such vulnerabilities exists and are common in organizations where different kind of threats are found in the organizations' infrastructure.

The problem is well recognized in the security community as well as in law-enforcement and intelligence communities. When prosecuting these kind of attacks the main measure to take is to audit log files and try to answer who had the motive and the possibility to commit the crime? Most common tool today is to analyse log files which faces the problem that the amount of data is overwhelming. In spite of this, software researchers have developed state-of-the-art models, automated tools, and techniques for analysing and solving (parts of) the problem.

System models are used to assess the vulnerability of information systems to security threats typically by representing a physical infrastructure (buildings) and a digital infrastructure (computers and networks), in combination with an attacker traversing the system while acquiring credentials. Presenting the environment formally (e.g. using mathematical notations) makes it possible to analyse potential insider attacks systematically and thus identify the set of suspects that could commit the crime.

1.1 Problem Statement

A number of system models have been introduced in order to support the analysis of organizations for their vulnerability to different kinds of threats by modelling the organizations infrastructure and actors. Examples of such models include ExASyM [PH08], Portunes [TD10] and ANKH [Pie11]. All these models are primarily based on similar ideas just with minor variations, namely the modelling of infrastructure and data, and analysing the modelled organization for possible attacks.

All these existing models are good at identifying various insider threats but are all composed such that there is no way to get any probabilistic timing information. This implies that it is currently not possible to get a quantitative probabilistic results for measuring how much risk there is for a certain attack will strike the organization and by whom with the current tools available.

This thesis aims to improve the existing models by combining them with more advanced analysis techniques so this additional probabilistic timing information can be obtained. The project will base its work on the ExASyM model (An **Ex**tensible **A**nalysable **Sy**stem **M**odel), and will apply the newly developed techniques and analysis for this model.

1.2 Thesis Organization

Chapter 2 introduce the ExASyM model on which this thesis is based on. The chapter will discuss some of the theoretical background that is necessary for following the rest of the thesis. The chapter will give an overview of the work done so far in analysing the insider problem and presents the extensions that have been made for this project.

Chapter 3 will give a brief introduction to model checking which is an automated technique that given a model of a system and a property, systematically checks whether this property holds for that model. In this chapter we will also discuss how to transform the ExASyM system model so it becomes suitable for model checking.

Chapter 4 will cover the some of the technical details and the implementation of the software tool that has been implemented for transforming the ExASyM model in to a LTS (tabelled transition system).

In Chapter 5 we will run the program on a few systems and do some analysis for measuring probabilities for existing threats. Finally Chapter 6 presents the conclusion of our work.

CHAPTER 2

ExASyM

This chapter gives an introduction to the ExASyM framework which makes the basis of this thesis. The goal of the chapter is to provide the reader with the necessary background to be able to follow the rest of the paper.
ExASyM got its name for being a system model that is highly extensible. The fundamentals of this framework is based on [CWPN06] which then has been extended with new components and more advanced analyses over the years in order to have a more realistic model of real-world systems. The details of these extensions can be found in [PH08, Gun07] and [PH09].
This chapter describe the latest version of the ExASyM model as it is now. Furthermore some new extensions and minor adaptations has been made for modelling purposes. These details are also covered in this chapter.

2.1 The Insider Problem

Just before presenting the framework, it is a good idea to discuss what is understood by an insider as this has recently been the topic of discussions. Often, insiders and outsiders are treated the same as they can cause the same damage once they have the same knowledge. However, they are and will be different with respect to the organization they attack. An insider is especially dangerous because even though he has information and capabilities not known to external attackers, he is trusted to perform some actions, which the outsider most certainly is not. Therefore their is a clear distinction between insiders and outsiders, where only the former can be source of insider threats. However, as a consequence of such a threat an outsider may obtain knowledge that will enable him to cause damage comparable to that an insider may cause.

The insider term is also closely related to "the insider threat". Bishop [Bis05] defines the term insider and insider threat as "*An insider with respect to rules R is a user who may take an action that would violate some set of rules R in the security policy, were the user is not trusted. The insider is trusted to take the action only when appropriate, as determined by the insider's discretion*". The insider threat "*is the threat that an insider may abuse his discretion taking actions that would violate the security policy when such actions are not warranted*".
A second definition found in the RAND report [RA05] defines the problem as "*malevolent actions by an already trusted person with access to sensitive information and information systems*", and the insider as "*someone with access, privilege, or knowledge of information systems and services*".

From these definitions it seems that the rules, R, consist of a set of access policies and a set of "trust" rules. The access granted to an actor could be limited to a special kind of circumstances, e.g., a janitor is not allowed to enter the server room in an organization unless there is fire in the room. However, the restriction does not really limit him to go there anyway, even if there is no fire. He is *trusted* not to go there under normal circumstances.

2.2 Informal System Description

ExASyM is a system model consisting of three important components namely location components, such as offices, rooms and computers, data components, such as keys and actual data, and mobile components, such as processes and actors. Data can be associated with (stored at) locations and actors, and it can be secured by, e.g., encryption, and locations can be secured by access control mechanisms, e.g., cipher locks. To support movements of dynamic components, locations are connected by directed edges, which define freedoms of movements of actors. Locations together with the connections between them compose the notation of an infrastructure. The model can contain different domains, which restricts the actors to only move in "their" domain, e.g., processes can only access the computer network.

The main property of the system model is that it is graph based and therefore lends itself to the application of simple graph-based analyses. While the underlying process calculus and its semantics can be used to analyse processes, which, e.g., could describe observed behaviour of an actor. Actors model everything that can move between locations and can perform actions. These actions consist of input, output, moving and evaluation, and are mapped to actions of acKlaim, a process calculus based on Klaim [RDNP98]. acKlaim is the underlying process calculus for ExASyM which provides formal semantics for the model and used in the analysis presented in section 2.11.

[PH08] present several extensions to the model, such as logging, encryption and decryption of data, and access control. The latter consist of annotations for locations that specify when access to the location is allowed. These annotations are evaluated by a reference monitor in the operational semantics before the corresponding action is performed, similar to [Gun07] and [CWPN06]. This is based on capabilities that actors can acquire, and restrictions that locations apply. Both restrictions and capabilities can be used to restrain the mobility of actors, by requiring, e.g., a certain key to enter a location, or allowing access only for certain actors, or from certain locations.

Comparing this informal model with real-world scenarios we are mostly concerned with office buildings and real data such as folders and documents, as well as computer networks and data that is available on these. The actors are the employees working at this office building which can perform some basic actions. All these are aspects are modelled and supported by the ExASyM system model.

2.3 Running Example

This section introduces a running example that will be used through out the report in order to have a clear understanding of how the introduced concepts work on a concrete example. The example is inspired by [Iva14] and is shown in Figure 2.1. The system consists of five locations, a reception which is the entrance to the office building, a janitor's workshop, an user's office with a computer and a printer, and the basement with a vault and a waste basket. A hallway is interconnecting the locations between the offices and the reception. Furthermore, the stairs in the hallway are leading to the basement. The computers and the printer are connected, so computers can send data to the printer and between each other. There are three actors in the system, a user, a janitor and a receptionist. Initially they are located at the office, at the janitor's workshop and at the reception respectively.

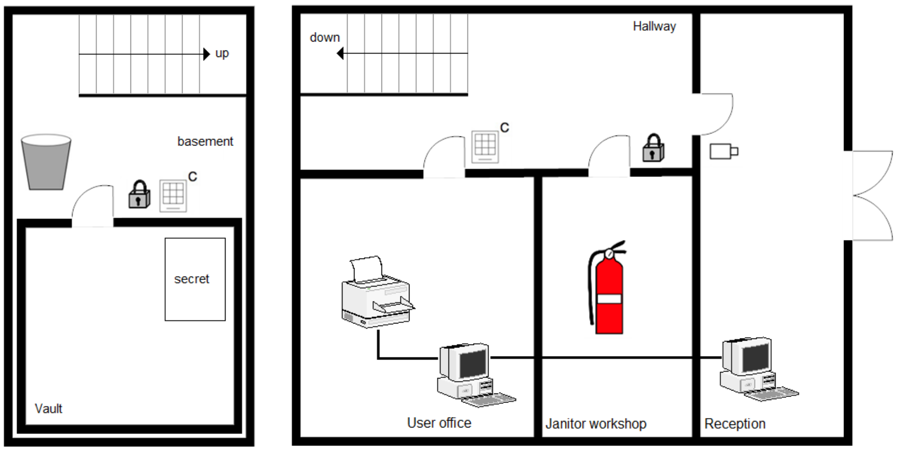


Figure 2.1: Running example

The user can use the network connection to print some confidential data on the printer and store them securely in the vault. Currently the vault stores a secret document. The hallway entrance is secured with a face recognition system.

The respective graph representation of the same model can be seen in Figure 2.2. This figure shows an abstraction of the real-world in which all rooms have become nodes in the graph, as have computers, the printer, the vault and the waste basket. In general, all elements of a system where data can be located are modelled as nodes. Additionally, places at which some kind of access control is applied can be turned into nodes. In the example these are all the doors (except the entrance). The connections between nodes are generated based on the type of connection they allow in the real system. Each location and computer has a policy annotation which restricts the actors freedom of actions and behaviour that they can perform on these peripherals.
The different kinds of arrows shown in Figure 2.2 indicate how connections can be accessed. These are the domains of the model specified in the abstract system. The solid lines, e.g., are accessible by actors modelling persons, the dashed lines by processes executing on the network. The dotted lines are special in that they express the interaction between a human person and a computer (an interaction over two different domains).

Actors in the system have to sign-in into the computer before they can use it, these constraints are modelled by policy annotations (marked in grey) on each computer in the system. For security reasons there is an cipher-locks on the entrance of the user's office. Each actor who wants to access the user's

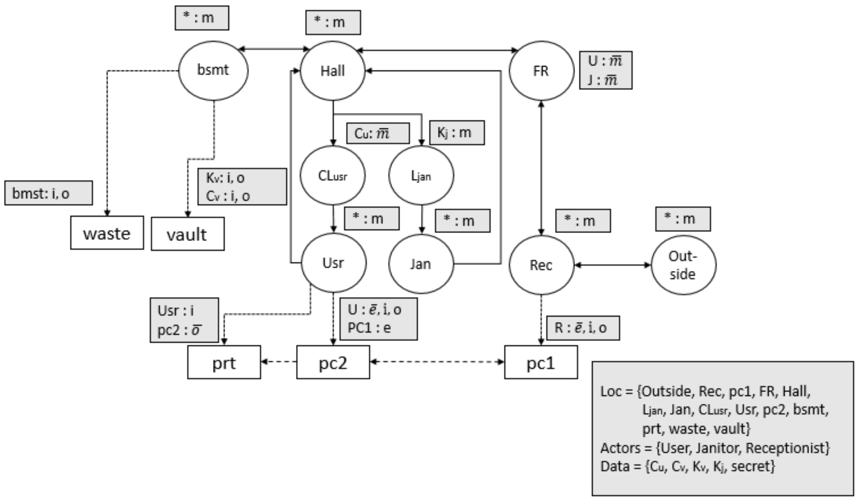


Figure 2.2: Abstraction for the example system shown in Figure 2.1

office has to know the PIN-code to be able to access the room. The janitor's workshop is locked with a regular lock, modelled in Figure 2.1 with a lock on the door. Therefore the janitor should have the corresponding key to open his own workshop. The vault has two keys installed on it. This does not mean that the actor who wants to access the vault has to have both keys. Only one of the keys should be represented in order to gain access to the vault. This is also shown in the policy restriction attached on it in Figure 2.2. Anyone who has one of these keys will be able to input or output data from the vault.

Data in this example could be data coming from the printer, the vault, or lying in the waste basket. The janitor also has the PIN-code to enter the vault, but it is assumed that he is "trusted" not to go into the vault unless there is something that needs to be fixed; e.g., repair a broken lock or in case of emergency (for instance in case of fire).

2.4 Insider Framework

Figure 2.3 gives an overview of the sequence of actions for performing insider analysis. The machinery takes the abstract, high-level system specification together with a log file as input and produces the results of the insider analysis.

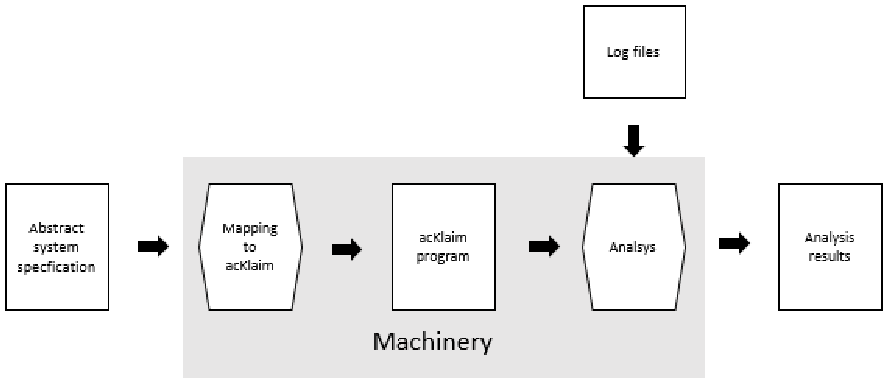


Figure 2.3: The insider framework assembly line

Inside the machinery the system specification is mapped to an acKlaim process calculus program and then the analysis on that program is being run. The reason for the mapping is that acKlaim has a formal semantics that is used to specify the analysis. At the final step of the assembly line, the results of the analysis is obtained.

2.5 Abstract System Model

The abstract system model is specified as a collection of mathematical constructs that is used to create an abstraction of a real-world system. Using these constructs makes it possible to model physical localities, interconnected computers, actors that can move around in the physical localities, and data that can be carried by actors or left at both computers or localities. It should be possible for actors to exchange data. On top of this there is a fine-grained access control mechanism that limits the mobility of actors, and protects sensitive data.

It should be noticed that in the abstract system, there is no means for modelling dynamic behaviours of actors, but only means of specifying what the initial structure of the system looks like - a static representation of the system model. The dynamic behaviour of the model is supported when the model is mapped to acKlaim in which the semantics of acKlaim uses the abstract system to evaluate the effects of actors movement.

The first definition is the notion of an infrastructure, which consists of a set of locations and connections. The infrastructure models the available connections in the modelled system, be it connections between rooms or computers, or be it the locks to access one location from another one, e.g. cipher-locks or face recognition.

DEFINITION 1: INFRASTRUCTURE, LOCATIONS AND CONNECTIONS

(Loc, Con) represents an infrastructure which is a directed graph, where Loc is a set of nodes representing locations, and $\text{Con} \subseteq \text{Loc} \times \text{Loc}$ is a set of directed edges between nodes representing connections between locations. Connections are used to define freedoms of movement for actors.
Two nodes are reachable if the nodes on the path is pair-wise connected to each other, more formally: $n_d \in \text{Loc}$ is reachable from $n_s \in \text{Loc}$ if there is a path $\pi = n_0, n_1, n_2, \ldots, n_k$ with $k \geq 1$ and $n_0 = n_s, n_k = n_d$ and $\forall_i \; 0 \leq i \leq k-1 : n_i \in \text{Loc} \wedge (n_i, n_{i+1}) \in \text{Con}$.
All locations are static and have no dynamic feature (as an elevator could have).
□

Actors are the dynamic entities in the system and can move along edges between nodes. Actors model persons moving around in the physical locations, or processes that migrate from one network location to the next.
It is therefore common to make a distinction between the “physical domain” and the “virtual domain” where actors move in their particular domain. PC’s are at the boundary between these two domains, as an actor could start a program on a computer which then migrates to another computer using the underlying network. Actors cannot move across domains boundaries, but data can do so as it might be uploaded to a computer, read from a screen or printed out. Data can be located at locations and actors, representing either what a user knows or possesses, e.g., by carrying around, or what data is available at a certain location. Data can be produced, picked up, and read by actors. The definition of data is rather loose, it can be any items that can be modelled as data at a convenient level of details. In the running example introduced in section 2.3, data could be used to model PIN-codes for the cipher locks or the key for the lock to enter the janitor’s workshop, Also printouts made from the workstations would be represented as data.
For this project we introduce a new element which we call for ELoc. ELoc is a location from which the actor entered the other domain (started a process on a computer). In this way we can make sure that an actor can not start more than one process on the same device. Each actor keeps information about this location. The notion will be further elaborated later in the report.

Definition 2: Actors and Domains

Let $\mathcal{I} = (\mathrm{Loc}, \mathrm{Con})$ be an infrastructure, Actors be a set. An actor $\alpha \in$ Actors is an entity that can move in $\mathcal{I}$ by following the edges between nodes.
Let Dom be a set of unique domain identifiers. Then $\mathcal{D} : \mathrm{Loc} \rightarrow \mathrm{Dom}$ defines the domain d for node n and $\mathcal{D}^{-1}$ defines all the nodes that are in a domain.
$\mathcal{G} : \mathrm{Actors} \rightarrow \mathrm{Loc}$ defines the location at which each actor is located and let $\mathrm{ELoc} : \mathrm{Actors} \rightarrow \mathrm{Loc}$ define the location at which the actor started a process in a different domain. □

Definition 3: Data

Let $\mathcal{I} = (\mathrm{Loc}, \mathrm{Con})$ be an infrastructure, Data is a set of data items and $\alpha \in$ Actors is an actor. A data item $d \in$ Data represent data available in the system. Data can be stored at both locations and actors.
$\mathcal{K} : (\mathrm{Actor} \cup \mathrm{Loc}) \rightarrow \mathcal{P}(\mathrm{Data})$ maps an actor or a location to a set of data stored at it. □

To model a system with access control, we need to model how actors can obtain the right to access locations, and how access to a location can be granted or denied. Access control is modelled with a set of capabilities and restrictions that restrain the mobility of the actors and protects sensitive data. Capabilities are associated with actors and restrictions are associated with locations and data. An actor can use his/her capabilities to something that enables him to get access to data or locations. Restrictions of data and locations are policies that limit access to a resource.

Definition 4: Capabilities and Restrictions

Let $\mathcal{I} = (\mathrm{Loc}, \mathrm{Con})$ be an infrastructure, Actors be a set of actors, and Data be a set of data items. Cap is a set of capabilities and Res is a set of restrictions. For each restriction $r \in$ Res, the checker $\Phi_r : \mathrm{Cap} \rightarrow \{true, false\}$ checks whether the capability matches the restriction or not.
$\mathcal{C} : \mathrm{Actors} \rightarrow \mathcal{P}(\mathrm{Cap})$ maps each actor to a set of capabilities and $\mathcal{R} : (\mathrm{Loc} \cup \mathrm{Data}) \rightarrow \mathcal{P}(\mathrm{Res})$ maps each location and data item to a set of restrictions. □

The extensions of ExASyM allows all actions to be logged. Each location or datum can now specify which actions on them are logged and which are not. This requires a log component to be defined. The logging component maps a log entry to the reason why an action was allowed or denied - that is a certain key, the actor's identity, or the location from which the request came from, as well as the location where the action was performed.

Definition 5: Log-component

Let T be a global system clock with type $\mathbb{N}$ which is a clock that enables to give each entry in the log file a unique label. The log file is modelled by a mapping: $\text{Log} : \mathbb{N} \times (\text{Actor} \cup \text{Data} \cup \text{Loc}) \times \text{Loc} \times \text{Loc} \times (\text{LocationAction} \cup \text{DataAction}) \rightarrow \mathcal{P}(\text{Res})$ which collects log file entries. The logged actions are added to a set of restrictions Res. Assume R is a set of restrictions, then the set of action modes is

$$\text{Res} = \bigcup \{r, \overline{r} | r \in \text{R}\}$$

where $\overline{r}$ is the logged action for r. □

We can now compose a system by combining the elements introduced above. Using locations, actors, data, and actions, it allows to capture the most important aspects of systems and insider threats, who the user is, what the user does and knows, and where the user does it.
Two systems are defined, in which one of them does not support any logging feature. Definition 6.A and 6.B gives a formal definition of these two systems.

Definition 6.A: Non-logged System

Let $\mathcal{I} = (\text{Loc}, \text{Con})$ be an infrastructure, Actors be a set of actors in $\mathcal{I}$, Data is a set of data items, $\mathcal{D} : \text{Loc} \rightarrow \text{Dom}$ a mapping from locations to domain, Cap is a set of capabilities, and Res is a set of restrictions. $\mathcal{C} : \text{Actor} \rightarrow \mathcal{P}(\text{Cap})$ is a mapping from actors to capabilities, $\mathcal{R} : (\text{Loc} \cup \text{Data}) \rightarrow \mathcal{P}(\text{Res})$ is a mapping from locations and data items to restrictions, $\mathcal{G} : \text{Actors} \rightarrow \text{Loc}$ is a mapping from actors to locations, and for each restriction $r \in \text{Res}$, let $\Phi_r : \text{Cap} \rightarrow \{true, false\}$ be a checker. ELoc : Actors $\rightarrow$ Loc maps actor's location at which the actor started a process in a different domain.
A non-logged system is defined as: $\mathcal{S} = \langle \mathcal{I}, \text{Actors}, \text{Data}, \mathcal{D}, \mathcal{G}, \text{ELoc}, \mathcal{C}, \mathcal{R}, \Phi \rangle$. □

Definition 6.B: Logged System

Let $\mathcal{I} = (\text{Loc}, \text{Con})$ be an infrastructure, Actors be a set of actors in $\mathcal{I}$, Data is a set of data items, $\mathcal{D} : \text{Loc} \rightarrow \text{Dom}$ a mapping from locations to domain, Cap is a set of capabilities, and Res is a set of restrictions. $\mathcal{C} : \text{Actor} \rightarrow \mathcal{P}(\text{Cap})$ is a mapping from actors to capabilities, $\mathcal{R} : (\text{Loc} \cup \text{Data}) \rightarrow \mathcal{P}(\text{Res})$ is a mapping from locations and data items to restrictions, $\mathcal{G} : \text{Actors} \rightarrow \text{Loc}$ is a mapping from actors to locations, and for each restriction $r \in \text{Res}$, let $\Phi_r : \text{Cap} \rightarrow \{true, false\}$ be a checker. ELoc : Actors $\rightarrow$ Loc maps actor's location at which the actor started a process in a different domain. Let T be a global clock with type $\mathbb{N}$ and $\text{Log} : \mathbb{N} \times (\text{Actor} \cup \text{Data} \cup \text{Loc}) \times \text{Loc} \times \text{Loc} \times (\text{LocationAction} \cup$

DataAction) $\to \mathcal{P}(\text{Res})$ be a logging component.
A logged system is defined as: $\mathcal{S} = \langle \mathcal{I}, \text{Actor}, \text{Data}, \mathcal{D}, \mathcal{G}, \text{ELoc}, \mathcal{C}, \mathcal{R}, \Phi, \text{T}, \text{Log} \rangle$.
□

2.6 Access Policies

Offering seamless access to computing resources and data from any location around the globe is one of the most outstanding features of modern computer systems. This flexibility makes them valuable, but at the same time is also the reason for their major vulnerability. Via the network, an entity's data is accessible from almost everywhere, often without the need of physical presence in the data's perimeter.
The risk of data being accessible without proper legitimation has led to a wide range of access control mechanism, which are supposed to restrict access to data. The standard approach to securing data is to tighten access control measures when these do not serve their purpose.

In order to limit access of actors to locations and data, a location or a datum has to explicitly grant access to an actor for performing a certain action on that datum or location.
ExASyM supports five different access modes (*i, o, e, m, d*) corresponding to the actions non-destructively read a tuple or pickup a tuple, output a tuple or produce a tuple. The *e* access mode is used to allow actors to spawn a new process in the system, and the *m* access mode controls the movement of actors into a neighbouring location from his current location. The *d* access mode is used for decrypting data items that actors are holding. It should be remarked that ExASyM does not support an access mode for encrypting data as it is unrealistic that a datum could make such a restriction to itself. Applying this approach affords fine-grained control for each individual localities and data items over both who is allowed access and how.

Every locality and data item in the system that defines an access policy specifies how actors are allowed to access and interact with it. Thus there are two access policies, one for locations called LocationPolicy and one for data called DataPolicy. These access policies are enforced by the reference monitor which is implemented in the semantics (process calculus). The reference monitor verifies that every access to a locality or a datum is in accordance with that locality's or datum's access policy. However the effect of each action is described by the semantics of the underlying process calculus (described in section 2.10). The access control policy for locations and data items is defined in Figure 2.4.
On the other hand it is also possible to not to specify any policy restrictions

$$\begin{aligned}
\text{UnloggedLocationActions} =& \{i, o, e, m\}\\
\text{LoggedLocationActions} =& \{\overline{i}, \overline{o}, \overline{e}, \overline{m}\}\\
\pi_\ell \subseteq \text{LocationAction} =& \text{UnloggedLocationActions} \cup \text{LoggedLocationActions}\\
\text{UnloggedDataActions} =& \{d, i, o\}\\
\text{LoggedDataActions} =& \{\overline{d}, \overline{i}, \overline{o}\}\\
\pi_\delta \subseteq \text{DataAction} =& \text{UnloggedDataActions} \cup \text{LoggedDataActions}\\
\kappa \subseteq \text{Data} =& \{\text{data items used as keys}\}\\
\delta \in \text{LocationPolicy} =& (\text{Loc} \cup \text{Actor} \cup \text{Data} \cup \{\star\}) \to R, R \in \mathcal{P}\{\text{LocationAction}\}\\
& \text{such that } \forall r \in R \Rightarrow \overline{r} \notin R \vee \forall \overline{r} \in R \Rightarrow r \notin R\\
\rho \in \text{DataPolicy} =& (\text{Loc} \cup \text{Actor} \cup \text{Data} \cup \{\star\}) \to R, R \in \mathcal{P}\{\text{DataAction}\}\\
& \text{such that } \forall r \in R \Rightarrow \overline{r} \notin R \vee \forall \overline{r} \in R \Rightarrow r \notin R
\end{aligned}$$

Figure 2.4: Access control policy for locations and data items

for a datum or for a location at all. In such case the datum or the location will be marked with an empty-set symbol $\emptyset$. This allows all possible actions to be performed on that datum or at that location. The actor is simply free to do whatever he wants to do as he wish. These locations or data items are called public.

The Data element in LocationPolicy and DataPolicy is used to model keys, so there is no distinction between data and keys. As soon as an actor picks up or reads a datum, it will be available to him as a capability.
Access modes in the set LocationAction are intended to be used for specifying access policies for locations only, while the access modes in the set DataAction are to be used for specifying access policies for data items.

The access to a location or a datum can be granted in three different way, based on...

- ... the identity of the actor performing the action (who).
- ... a secret (key/datum) known by the actor (what).
- ... the location of the actor where the access request is coming from (where).

Actions that are marked with an over line are logged actions that cause a log-entry in the log-component. The access control policy also ensures that a policy

does not contain a logged action along with its non-logged counterpart.
The special character $\star$ allows to specify a set of access modes that are allowed by default for all actors in the system.

2.7 System Specification Language

The abstract system specification described so far is not well suited for implementation as it is all been based on a collection of mathematical definitions. Therefore a system specification language has been introduced for specifying a system model in text files, which then can be used as input for the analyses. Although an abstract specification will be mapped to an acKlaim program, the syntax of the language has been designed to be as close to the abstract specification as possible with the intention that the user should not have to know anything about the underlying theory in order to use the analyses tool.

The system specification language is shown in Figure 2.5 and Figure 2.6. The system is composed of four major categories: locations, connections, actors, and data. Locations are represented by a location name along with a list of restrictions that the location makes on actions performed on it. Each restriction is a name specifying a location, an actor, a piece of data, or a $\star$ (star) representing a wild card, and a list of actions that the given name is allowed to perform at the location. Each location also specifies the domain it is member of. The domain is simply a name and must not conflict with names used for other purposes. The list of restrictions for a location may be empty, meaning that no restrictions are imposed on the access of that location (public location). To model that no access is allowed by anyone, the list of access modes should be left empty likewise: `LocationName{⋆ :} DomainName`. Be aware of that in cases where the restrictions are totally left empty (`LocationName{} DomainName`), the actor will be able to perform any action he wants on that location. The same is also applicable for data items.

Connections are directed edges specified with a right-pointing arrow from a location A to other locations B, C, D,... , meaning that there is an edge from A to all the other locations. Both end points in a connection must be defined for the connection to be well-formed.

Actors are represented by a list of pairs of actor names and the name of the location(s) the actor is initially located. Note that, in the case of uncertainty, an actor may be placed at several locations. The set of actor names must be disjoint from the set of location names for the specification to be well-formed.

$$
\begin{aligned}
\mathit{Spec} ::=& \texttt{locations} : \{\mathit{Locations}\}\\
& \texttt{connection} : \{\mathit{Connections}\}\\
& \texttt{actors} : \{\mathit{Actors}\}\\
& \texttt{data} : \{\mathit{Data}\}\\
\mathit{Locations} ::=& \mathit{Location} *\\
\mathit{Location} ::=& \texttt{LocationName}\ \{\mathit{LocationPolicies}\}\ \texttt{(DomainName);}\\
\mathit{LocationPolicies} ::=& \mathit{LocationPolicy} *\\
\mathit{LocationPolicy} ::=& \mathit{Name} : \mathit{AllowedActions};\\
& \mid \star : \mathit{AllowedActions};\\
\mathit{AllowedActions} ::=& \mathit{LocationAction}?\ (,\ \mathit{AllowedActions})^*\\
\mathit{LocationAction} ::=& \texttt{i} \mid \texttt{ilog} \mid \texttt{o} \mid \texttt{olog} \mid \texttt{e} \mid \texttt{elog} \mid \texttt{m} \mid \texttt{mlog}
\end{aligned}
$$

Figure 2.5: System Specification Language, part 1

$$
\begin{aligned}
\mathit{Connections} ::=& \mathit{Connection} *\\
\mathit{Connection} ::=& \texttt{LocationName -> } \mathit{LocationNames};\\
\mathit{Actors} ::=& \mathit{Actor} *\\
\mathit{Actor} ::=& \texttt{ActorName @ } \mathit{LocationNames};\\
\mathit{Data} ::=& \mathit{Datum} *\\
\mathit{Datum} ::=& \texttt{DataName}\ \{\mathit{DataPolicies}\}\ \texttt{@ LocationName;}\\
& \mid \texttt{DataName}\ \{\mathit{DataPolicies}\}\ \texttt{@ ActorName;}\\
\mathit{DataPolicies} ::=& \mathit{DataPolicy} *\\
\mathit{DataPolicy} ::=& \mathit{Name} : \mathit{DataAction}?;\\
& \mid \star : \mathit{DataAction}?;\\
\mathit{DataAction} ::=& \texttt{d} \mid \texttt{dlog} \mid \texttt{i} \mid \texttt{ilog} \mid \texttt{o} \mid \texttt{olog}\\
\mathit{Name} ::=& \texttt{ActorName} \mid \texttt{DataName} \mid \texttt{LocationName}\\
\mathit{LocationNames} ::=& \texttt{LocationName}\\
& \mid \texttt{LocationName, } \mathit{LocationNames}
\end{aligned}
$$

Figure 2.6: System Specification Language, part 2

The location at which an actor is located must also be defined in the list of locations.

Data is specified as a list of data elements annotated with access restrictions and information on where they are located (either an actor holds it or it is placed in a location). For ease of presentation, the structure of data is one-dimensional single string. This can easily be extended to a more complex tuple structure with nested tuples and so on. It should be noted that such a change does not require changes to the techniques presented here. If the list of restrictions is empty, the datum is assumed to be public, and if the list of access modes is empty for the $\star$ element, the datum cannot be accessed by any actor in the system.
Access restriction on data may be decryption read and write restrictions, modelled as input and output. Any actor is free to pick up or read data (as long as he has access to it), but to get the information that encrypted data holds, he needs to have the corresponding key to be able to decrypt it.

The source code in Figure 2.7 shows the representation of the example system for Figure 2.1. Beyond the system graph shown there, the textual representation also includes the data available at actors or locations, as well as the locations where actors are located initially. In order to run through different scenarios, these locations, as well as the data available, can easily be changed.

2.8 acKlaim Syntax

Distributed systems typically consist of a large number of heterogeneous computational entities that execute components of applications. Components of distributed systems have to deal with unpredictable changes in the network environment over time, e.g., mobile components can disconnect from the network and reconnect later at a different node.
Klaim (**K**ernel **L**anguage for **A**gents **I**nteraction and **M**obility) is a language designed to model distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces.

Klaim is used for modelling processes running in a wide area network, where the structure of the network can change. Processes in Klaim can change the spatial structure of the network, and localities are first-class citizens that can be dynamically created and communicated over the network. A tuple space is a multi-set of tuples that are sequences of information items. Tuples are anonymous and picked up from tuple spaces by means of pattern-matching. Two tuples match if they have the same number of fields and corresponding fields

```
locations:{
  Outside {⋆:  m;} (building);
  Rec {⋆:  m;} (building);
  pc1 {R: elog, i, o;} (virtual);
  FR {U: mlog; J: mlog;}
     (building);
  Hall {⋆:  m;} (building);
  L_jan {K_j:  m;} (building);
  Jan {⋆:  m;} (building);
  CL_usr {C_u:  mlog;} (building);
  Usr {⋆:  m;} (building);
  pc2 {U: elog, i, o; pc1:  e;}
     (virtual);
  prt {Usr:  i; pc2:  olog;}
     (device);
  Bsmt {⋆:  m;} (building);
  vault {K_v:  i, o; C_v:  i, o;}
     (inventory);
  waste {Bsmt:  i, o;} (inventory);
}
connections:  {
  Outside -> Rec;
  Rec -> pc1, FR, Outside;
  pc1 -> pc2;
  FR -> Hall, Rec;
  Hall -> FR, L_jan, CL_usr, Bsmt;
  L_jan -> Jan;
  Jan -> Hall;
  CL_usr -> Usr;
  Usr -> Hall, pc2, prt;
  pc2 -> pc1, prt;
  Bsmt -> Hall, waste, vault;
}
actors:{
  U @ Usr;   J @ Jan;
  R @ Rec;
}
data:{
  C_u@ U;   C_v@ U;
  K_j@ J;   K_v@ J;
  secret{} @ vault;
}
```

Figure 2.7: Running example modelled in the System Specification Language

have matching values or variables. The interprocess communication between the processes is asynchronous; sender and receiver does not need to synchronize their actions beforehand.

Klaim is a family of process calculi and the most basic version is cKlaim or Core Klaim, which can be seen as a variant of the π-calculus. μKlaim (micro Klaim), is an extension of cKlaim with tuples and pattern matching. acKlaim, which is used in the ExASyM project, is an extension of μKlaim which is enhanced with access control primitives and equipped with a reference monitor semantics that ensures compliance with the system's access policy. There exists also many other extensions to cKlaim which among others can be found in [BBN$^+$03, HPN06, GP03] and [RDNP98], but shall not be covered here.

This section present the syntax of acKlaim process calculus. The abstract system modelled in section 2.7 will be mapped to a acKlaim program on which the analysis is applied, since acKlaim provides a formal semantics.
The syntax of acKlaim is similar to the μKlaim syntax, in which the main difference to standard Klaim calculi is that processes are annotated with a name, in order to model actors moving in a system, and a set of keys to model the capabilities they have. The syntax consists of four syntactic categories: localities, nets, processes and actions, as shown in Figure 2.8 and Figure 2.9.

Localities can be a name (actor) or a locality variable (location). The locality variable can be used to communicate a name of a location between actors, e.g., actors could decide to meet at a given location and one of the actors communicates to the other actor by giving him a name of a location. Actors communicate by outputting data, picking data up, or reading it.

Nets are finite collections of nodes that represent the localities in the system and can contain processes and data. A node can be referenced by its locality, l, which represents the address of the node, and is either a process or a datum. If the node is a process node, the node is annotated with a LocationPolicy, the name of an actor, and the actors keys. On the other hand, if the node is a datum node, it is annotated with a policy from DataPolicy, and contains a datum element. Finally, nets can be the composition of two nets. Note that nets do not give any information on how nodes are interconnected.

Processes are the active computational units of acKlaim and are executed by performing sequences of basic operations over the tuple space and nodes. A process can run concurrently in composition of two or more processes at either the same location or at different locations in the system, and can contain an invocation to a named process definition.
A process can also be the **nil** process that simply does nothing, or a sequence of actions built up from the **nil** process. A locality containing the **nil** process

models a location with no actor and no data.

There are six actions provided in the acKlaim syntax. The **out** action produces a tuple and places it at a given location. The **in** action non-destructively reads a tuple from the specified location. The location accessed by the **out** and **in** action is from the current location of the actor, or one of its neighbouring locations that is in a different domain (e.g. office –> pc). As a result an actor cannot magically output or read a datum from a location that is not "close" to him.

The **eval** action is used for actors to spawn a new process in a different domain from the actor's current location. In such case the ELoc will be set to the location where the **eval** action happened, such that the actor who created the process can not spawn a second process on the same device (The ELoc represent the physical location where the actor is located). In real-life situation there wouldn't be such a restriction and the number of processes the actor could create would in the theory be infinity. However this restriction has been needed for this project as the number of processes (actors) in the system could grown to infinity which would blown up the of state-space in the generated LTS, as the number of states is depending on the number or actors presented in the system (section 3.7).
The created process in the other domain can now migrate to other locations in the same domain or in other domains. In this way programs executing on computers can move themselves to other computers and resume its execution while actors in the physical domain can move to other locations as well. These constraints will be enforced by the semantics (in section 2.10). The movement of actors is realized by the **move** action, that moves the actor to the specified location.

The **encrypt** and **decrypt** actions are special actions that an actor can perform to encrypt and decrypt data. Any data can be encrypted as long as the actor has access to the data. The current model of ExASyM does not allow encryption of data that has already been encrypted, and thus create a chain of encryption on the data. This is due to the security restrictions applied for data items which would need to be more flexible than they are now. The set of security restrictions would have to be a list of security restrictions where the ordering of restrictions would matter when decrypting the datum such that the decryption ordering match with the encryption ordering.
Decryption is only possible if the data allows the actor to perform the decryption, either through his location, his identity, or his keys. As mentioned before decryption is the only access restriction a datum can make, as data can be picked up and moved by any actor.

$\ell ::= l$	locality	$P ::= \mathbf{nil}$	null process
$\mid u$	locality variable	$\mid a.P$	action prefixing
		$\mid P_1 \mid P_2$	parallel composition
		$\mid A$	process invocation
$N ::= l ::^{\delta} [P]^{\langle n,\kappa\rangle}$	single node	$a ::= \mathbf{out}(t)@\ell$	output
$\mid l ::^{\delta} \langle et^{\rho}\rangle$	located tuple	$\mid \mathbf{in}(T)@\ell$	input
$\mid N_1 \parallel N_2$	net composition	$\mid \mathbf{encrypt}(t, \rho, F)$	encrypt
		$\mid \mathbf{decrypt}(t, F)$	decrypt
		$\mid \mathbf{eval}(P)@\ell$	migration
		$\mid \mathbf{move}(l)$	move

Figure 2.8: acKlaim syntax of nets, processes and actions

$\mathrm{T} ::= \mathrm{F} \mid \mathrm{F},\ \mathrm{T}$	templates	$et ::= ef \mid ef, et$	evaluated tuple
$\mathrm{F} ::= f \mid !x \mid !u$	template fields	$et ::= ef \mid ef, et$	evaluated tuple fields
$t ::= f \mid f, t$	tuples	$e ::= V \mid x \mid \ldots$	expressions
$f ::= e \mid l \mid u$	tuple fields		

Figure 2.9: Syntax for tuples and templates

Tuples are the communicable objects in acKlaim and are sequences of actual fields. The actual fields are expressions, localities, and locality variables. The expressions are deliberately not specified but contain at least values, V, and value variables, x. Templates are sequences of actual and formal fields. Formal fields are variables with an exclamation mark like (!x, !u) are used to bind values.
When tuple spaces are read by using either **in** a tuple that matches the input pattern (template) is read in an arbitrary way. For actions **in** and **out** the templates must be evaluated before they are added to a tuple space. The template evaluation consists of computing the value of the expressions occurring in the template. Templates with variables in actual fields cannot be evaluated. The $[\![\mathrm{T}]\!]$ is used to denote an evaluated template.

2.9 Mapping Systems to acKlaim

The acKlaim syntax is used to describe a system model as the one specified in Figure 2.7. The resulting acKlaim program for the running example is shown in Figure 2.10. The system property that is most interested is the initial structure of the system, therefore most locations run either the **nil** process or have an empty tuple space. The receptionist, user's office and the janitor's workshop contain process variables to model the actors at their initial location in the system.

The transformation from the specification language to the acKlaim syntax is a quite simple procedure. The procedure iterates over each location defined in the location section of the specification and creates a new process node in the resulting net. Each node gets the same name and access control attribute as in the specification language and a **nil** process. At locations where an actors is present the process will be a process variable that is named with the name of the actor and has the actor's keys available. There is no way to specify the spacial structure of a system with the acKlaim syntax, so the connections of the specification language cannot be mapped to anything useful.

$$
\begin{array}{ll@{\qquad}ll}
\mathrm{Outside} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathbf{nil}] & \| & \mathrm{Usr} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathrm{U}]^{\langle U,\ \{\mathrm{C_u},\mathrm{C_v}\}\rangle} & \| \\
\mathrm{Rec} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathrm{R}]^{\langle R,\ \{\}\rangle} & \| & \mathrm{pc2} ::^{\langle \mathrm{U}\mapsto \mathrm{elog,\ i,\ o,pc1}\mapsto \mathrm{e}\rangle} [\mathbf{nil}] & \| \\
\mathrm{pc1} ::^{\langle \mathrm{R}\mapsto \mathrm{elog,\ i,\ o}\rangle} [\mathbf{nil}] & \| & \mathrm{prt} ::^{\langle \mathrm{Usr}\mapsto \mathrm{i,pc2}\mapsto \mathrm{olog}\rangle} [\mathbf{nil}] & \| \\
\mathrm{FR} ::^{\langle \mathrm{U}\mapsto \mathrm{mlog},\ \mathrm{J}\mapsto \mathrm{mlog}\rangle} [\mathbf{nil}] & \| & \mathrm{Bsmt} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathbf{nil}] & \| \\
\mathrm{Hall} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathbf{nil}] & \| & \mathrm{vault} ::^{\langle \mathrm{K_v}\mapsto \mathrm{i,\ o,C_v}\mapsto \mathrm{i,\ o}\rangle} [\mathbf{nil}] & \| \\
\mathrm{L_{jan}} ::^{\langle \mathrm{K_j}\mapsto \mathrm{m}\rangle} [\mathbf{nil}] & \| & \mathrm{vault} ::^{\langle \mathrm{K_v}\mapsto \mathrm{i,\ o,C_v}\mapsto \mathrm{i,\ o}\rangle} \langle \mathrm{secret}^{\langle \emptyset\rangle} \rangle & \| \\
\mathrm{Jan} ::^{\langle \star\mapsto \mathrm{m}\rangle} [\mathrm{J}]^{\langle J,\ \{\mathrm{K_v},\mathrm{K_j}\}\rangle} & \| & \mathrm{waste} ::^{\langle \mathrm{Bsmt}\mapsto \mathrm{i,\ o}\rangle} [\mathbf{nil}] & \\
\mathrm{CL_{usr}} ::^{\langle \mathrm{C_u}\mapsto \mathrm{mlog}\rangle} [\mathbf{nil}] & \| & &
\end{array}
$$

Figure 2.10: The example system translated into acKlaim progam

2.10 Semantics of acKlaim

The operational semantics of acKlaim language is defined by a structural congruence, a reference monitor and a set of reduction relation. The structural congruence incorporates the basic semantics of nets in parallel composition, while the reference monitor checks whether access to data and localities are in accordance with the access policy defined for it. The reduction relations describes the basic computational paradigm of interactions among processes inside a net.

2.10.1 Congruence Relation

The structural congruence $\equiv$ identifies, which nets intuitively represent the same net. The structural congruence relation simplifies presentation of the semantics and the reasoning about processes. The relation is defined in Figure 2.11.

$$
\begin{array}{ll}
(\mathit{Com}) & N_1 \parallel N_2 \equiv N_2 \parallel N_1 \\
(\mathit{Assoc}) & (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \\
(\mathit{Abs}) & l ::^{\delta} [P]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [(P \mid \mathbf{nil})]^{\langle n,\kappa \rangle} \\
(\mathit{Inv}) & l ::^{\delta} [A]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [P]^{\langle n,\kappa \rangle} \;\; \text{if } A \triangleq P \\
(\mathit{Clone}) & l ::^{\delta} [(P_1 \mid P_2)]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [P_1]^{\langle n,\kappa \rangle} \parallel l ::^{\delta} [P_2]^{\langle n,\kappa \rangle}
\end{array}
$$

Figure 2.11: Structural congruence on nets and processes

The (Com) rule represents the commutative law for nets. Since the net does not give any explicit structure of nodes, it does not matter in which order they are presented in the net. The (Assoc) rule represents the associativity law for nets, which means that the order of evaluation does not matter, as all processes in the net run concurrently. The (Abs) rule says that it is safe to add/remove a concurrent running **nil** process to a process P and not result in a new net. The (Inv) rule represents process invocation and says that when invoking a process named A, the body of the process A, is substituted out for the name, if there exists a process definition for A. The (Clone) rule says that a single process with two concurrent running parts in a single node can be split into two nodes, running each part separately. This will not create a different net, as the two process parts will be running at the same location as before, with the same access restrictions.

2.10.2 Reference Monitor

The access control mechanism in ExASyM is enforced by a reference monitor that is embedded in the operational semantics of the process calculus. It checks whether access to data and localities are in accordance with the policy of the datum or locality and whether the actor is allowed to perform a certain action before he actually performs it.
The reference monitor has several parts and is defined in Figure 2.12 and in Figure 2.13.

In order to support the semantics for the log-file extension, an additional global timer T and the logging component Log has been added. It has been aimed for designing a logging extension to model a realistic log file as possible by logging what kind of information that would be available at the given location or data item. It is possibly to log information such as actor's name, the action he wants perform from a location to a target location, and of course the timestamp as all these information is available in each semantic rule. But this would not be realistic for all scenarios for logged actions. For instance in the scenario where an actor uses a PIN-code to move to a location, in reality a cipher lock would probably only register the time and possibly the key used for the access, as the lock would have no way of knowing which actor was granted access.
The logging extension works by recording by which means the access was granted. For this purpose two helper functions, `grantby` and `decryptby` are introduced, listed in Algorithm 1 and Algorithm 2. These two algorithms are used to provide the semantics with information on how access was granted.

1. Access can be granted based on the identity of the actor or that access was not restricted at all (e.g. the access policy was defined with the $\star$ or $\emptyset$ element),
2. Loc represents that access was granted based on the current location of the actor,
3. Data represents that access was granted based on some knowledge of the actor,
4. ϵ represent that access was not granted at all.

Notice that in cases where either a policy is defined with the $\star$ element or a resource is unrestricted, the functions `grantby` and `decryptby` returns the identity of the actor that is granted the access for.

For this project we have added an additional function called `checkdomain`, Algorithm 3, to the reference monitor. The function restrains some of the actions

Algorithm 1 grantby(Names $\times$ Loc $\times$ $\mathcal{P}$(Data) $\times$ LocationAction $\times$ Loc) $\rightarrow$ (Names $\times$ Data $\times$ Loc $\times$ $\{\epsilon\}$)

```
function GRANTBY($n, l, \kappa, a, l'$)
    if $\delta_{l'} = \emptyset$ then
        return $n$                         ▷ no access control on $l'$ (public location)
    else if $\star \in \delta_{l'}^{-1}(a)$ then
        return $n$                         ▷ unrestricted access for all actors (star-property)
    else if $a \in \delta_{l'}(n)$ then
        return $n$
    else if $a \in \delta_{l'}(l)$ then
        return $l$
    else if $\exists k \in \kappa : a \in \delta_{l'}(k)$ then
        return $k$
    else
        return $\epsilon$                  ▷ access denied
    end if
end function
```

Algorithm 2 decryptby(Names $\times$ Loc $\times$ $\mathcal{P}$(Data) $\times$ DataAction $\times$ Data) $\rightarrow$ (Names $\times$ Data $\times$ Loc $\times$ $\{\epsilon\}$)

```
function DECRYPTBY($n, l, \kappa, a, et^{\rho}$)
    if $\rho_{et} = \emptyset$ then
        return $n$                         ▷ public data with no access control
    else if $\star \in \rho_{et}^{-1}(a)$ then
        return $n$                         ▷ public data for all actors (star-property)
    else if $a \in \rho_{et}(n)$ then
        return $n$
    else if $a \in \rho_{et}(l)$ then
        return $l$
    else if $\exists l' \in \{l'' \mid (l, l'') \in \text{Con} \wedge a \in \rho_{et}(l'')\}$ then
        return $l'$
    else if $\exists k \in \kappa : a \in \rho_{et}(k)$ then
        return $k$
    else
        return $\epsilon$                  ▷ decryption failed
    end if
end function
```

that an actor can perform. For instance it should not be possible to input or output data between two locations that are in the same domain. This would be like throwing or pulling a data item between two locations. Therefore an actor who want to output or input a data item has to be at that location where the data item is located. However if the two locations are in different then all action that are allowed can be performed. E.g. the actor does not need to go inside the vault to input a data item. Also all permissible actions that the actor can perform without moving to another location are allowed.

Algorithm 3 checkdomain(Loc × Loc × (LocAction ∪ DataAction) → Boolean)

```
1: function CHECKDOMAIN($l, l', a$)
2:     if $l \neq l' \wedge \text{dom} = \text{dom}(l') \wedge a \in \{m, mlog, e, elog\}$ then
3:         return $true$
4:     else if $l = l' \vee \text{dom} \neq \text{dom}(l')$ then
5:         return $true$
6:     else
7:         return $false$
8:     end if
9: end function
```

(1)

$$grant : \text{Names} \times \text{Loc} \times \mathcal{P}(\text{Data}) \times \text{LocationAction} \times \text{Loc} \to \{true, false\}$$

$$grant(n, l, \kappa, a, l') = \begin{cases} true & \text{if } grantby(n, l, \kappa, a, l') \neq \epsilon \\ false & \text{otherwise} \end{cases}$$

(2)

$$\frac{l = l' \vee \exists (l, l') \in \text{Con}}{\langle \mathcal{I}, n, \kappa \rangle \succ (l, l')}$$

(3)

$$\frac{grant(n, l, \kappa, a, l') \wedge \langle \mathcal{I}, n, \kappa \rangle \succ (l, l') \wedge checkdomain(l, l', a)}{\langle \mathcal{S}, n, \kappa \rangle \rightsquigarrow \langle l, l', a \rangle}$$

Figure 2.12: Reference Monitor for access control, part 1

The *grant* function (1) in Figure 2.12 takes an actor n, a location l, where the actor is currently located, a set of keys κ, the action a that the actor wants to perform and the target location l' where the action has to be performed as parameters. It checks that the actor has access rights to perform action a at the specified location l' by using the `grantby` function. The *grant* function does not check whether the actor can actually reach location l' from location l. The type of the a is LocationAction as defined in section 2.4. The *grant* function is used to define the judgement in (3).

The $\succ$ judgement (2) specifies the condition that has to hold for an actor to be able to reach location l' from l. Either the actor has to be in location l' or one of the neighbouring location to l'.

Finally we have the $\rightsquigarrow$ judgement which is the reference monitor for location access. It is composed of rule (1), (2) and `checkdomain`. It specifies the conditions that has to hold for an actor n to be able to perform the action a at location l' when he is located at l. The condition that must hold are that the actor is located in a neighbouring location to l' and have access rights to perform action a at location l'.

(4)

$$\mathit{decrypt} : \text{Names} \times \text{Loc} \times \mathcal{P}(\text{Data}) \times \text{DataAction} \times \text{Data} \rightarrow \mathcal{P}(\text{Data})$$

$$\mathit{decrypt}(n, l, \kappa, a, et^{\rho}) = \begin{cases} \{et^{\emptyset}\} & \text{if } \mathit{decryptby}(n, l, \kappa, a, et^{\rho}) \neq \epsilon \\ \emptyset & \mathit{otherwise} \end{cases}$$

(5)

$$\mathit{encrypt} : \text{Names} \times \text{Loc} \times \mathcal{P}(\text{Data}) \times \text{Data} \times \text{DataPolicy} \rightarrow \mathcal{P}(\text{Data})$$

$$\mathit{encrypt}(n, l, \kappa, et^{\rho}, \rho') = \begin{cases} \{et^{\rho'}\} & \mathit{decrypt}(n, l, \kappa, \mathrm{d}, et^{\rho}) = \{et^{\emptyset}\} \\ \emptyset & \mathit{otherwise} \end{cases}$$

Figure 2.13: Reference Monitor for access control, part 2

The rules in Figure 2.13 defines the access control for encryption and decryption of data items. The rules are used in the reduction relations in next section (section 2.10.3).

The *decrypt* function (4) specifies the condition that has to hold for an actor n located at l, holding a set of keys k, to be able to decrypt the datum et, which is encrypted with ρ. The function returns the decrypted datum, with no access restriction attached to it or an empty set, if access to the datum is denied.
The *decrypt* function is applied to data that the actor is in possession of. This means that he cannot decrypt data located in other localities or at other actors. The result of the decryption depends on the `decryptby` function which basically checks whether the actor can access the datum.

The *encrypt* function (5) specifies the condition that has to hold for an actor n, located at l, with keys k to be able to encrypt the data et^{ρ} with the new key ρ'. Only public data can be encrypted, and it is not possible to make a chain of encryptions on a data item.

The function checks if the actor is able to decrypt the datum, if so the security annotation of the datum is replaced with ρ'.

2.10.3 Reduction Relations

The operational semantics of nets exploits an evaluation mechanism for tuples and a pattern–matching to select tuples in a tuple space. The semantics for acKlaim (shown in Figure 2.14, 2.15 and 2.17) is specified in a small step operational semantics and follows the semantics of μKlaim quite closely.

The (IN) rule says that an actor performing the **in** action reads data from location l' (it does not remove the data item from the location!). The template T is used as a pattern to find which data at l' to retrieve. The match function creates a substitution, and the retrieved data will be substituted out for the reference to it in the resulting process. The actor will of course have to have the access to perform the **in** action. The (IN) rule uses the reference monitor semantics to check that the actor is in the right location, and that he has access to perform the **in** action on that location. When data is picked up from the location l', the $\mathcal{K}$ component of $\mathcal{S}$ is updated to reflect that the data has been added to the given actor's key set. Even though an actor has "consumed" an encrypted data using the (IN) rule, it does not mean that he can decrypt it and understand it. Although the user cannot understand the datum, it is added to his key set. When the user performs a successful decrypt action on the encrypted datum the decrypted datum will be available as a key to him and added to his key set κ.

The (OUT) rule says that if an actor, located at l, performs the **out** action in location l' the datum will be added to the tuple space of l', if the following three conditions hold:

1. the location l' must exist in the abstract system,
2. the security annotation of l' is δ' and
3. t evaluates to et^{ρ}.

Furthermore the actor n has the required access to perform the **out** operation on location l'. The $\rightsquigarrow$ relation ensures that the actor is either located at l' or in a neighbouring location (that is in a different domain).

In this project we have extended the (EVAL) rule in order to reduce the number of actors (processes) that can be created in the system. This has been necessary as the state-space of the resulting LTS could become very large. Basically what we have done is that we allow actors to move into the other domain, but at the

$$\text{(IN)}\;\frac{match([\![T]\!], et^{\rho}) = \sigma \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathrm{i}\rangle} \qquad \mathcal{K}' = \mathcal{K}[n \to \mathcal{K}[n] \cup \{et^{\rho}\}]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{in}(T)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa\cup\{et^{\rho}\}\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle}$$

$$\text{(OUT)}\;\frac{\begin{array}{ccc} l' \in \mathrm{Loc} & \mathcal{R}(l') = \delta' & [\![t]\!] = et^{\rho} \\ \multicolumn{3}{c}{\mathcal{K}' = \mathcal{K}[l' \to \mathcal{K}[l'] \cup \{et^{\rho}\}] \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathrm{o}\rangle}}\end{array}}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{out}(t)@l'.P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle}$$

$$\text{(EVAL}_{\mathbf{nil}}\text{)}\;\frac{l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathrm{e}\rangle} \qquad \mathrm{ELoc}' = \mathrm{ELoc}[n \to l]}{\begin{array}{c}\mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(Q)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \rightarrowtail \\ \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle}\end{array}}\;\mathrm{ELoc}(n) = \mathbf{nil}$$

$$\text{(EVAL}_{\mathbf{def}}\text{)}\;\frac{l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathrm{e}\rangle}}{\begin{array}{c}\mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(Q)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \rightarrowtail \\ \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle}\end{array}}\;\mathrm{ELoc}(n) \neq \mathbf{nil}$$

$$\text{(MOVE}_{\mathbf{diff}}\text{)}\;\frac{\begin{array}{cc} l' \in \mathrm{Loc} & \mathcal{R}(l') = \delta' \\ \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathrm{m}\rangle} & \mathcal{G}' = \mathcal{G}[n \to l']\end{array}}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle}}\;\mathrm{Dom}(l) \neq \mathrm{Dom}(\mathrm{ELoc}(n))$$

Figure 2.14: Operational Semantics for acKlaim (unlogged actions), part 1a

same time remember the location where he entered the other domain such that he can always move in a nearby location where the **eval** action was executed. In order to do this we have introduced two semantic rules that describes how to spawn a new processes. Depending on the side condition, one of them will be evaluated.
The (EVAL$_{\mathbf{nil}}$) describes the **eval** action when the actor is in the domain he initially started. In that case the ELoc of n will be **nil**. From the viewpoint of the resulting net the actor will be moved into the other domain, while the rule makes sure to set the ELoc of n to l, such that the actor will also be able to move to other locations that is reachable from ELoc, even if he moves away from l' in the long run.
The (EVAL$_{\mathbf{def}}$) describes the rule when the actor want to spawn a process when ELoc is set. The process can spawn a new process in another location,

$$(\text{MOVE}_{\mathbf{eq}})\dfrac{l' \in \text{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \mathcal{G}' = \mathcal{G}[n \to l'] \\ \boxed{\langle \mathcal{S}, n, \kappa \rangle \leadsto \langle l, l', \text{m} \rangle} \qquad \text{ELoc}' = \text{ELoc}[n \to \mathbf{nil}]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa \rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa \rangle}}\text{Dom}(l) = \text{Dom}(\text{ELoc}(n))$$

$$(\text{ENCRYPT})\dfrac{\mathit{encrypt}(n, l, \kappa, [\![t]\!], \rho) = \{et^{\rho}\} \qquad \mathit{match}([\![F]\!], et^{\rho}) = \sigma \qquad \mathcal{R}' = \mathcal{R}[et \to \rho]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{encrypt}(t, \rho, F).P]^{\langle n,\kappa \rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup \{et^{\rho}\} \rangle}}$$

$$(\text{DECRYPT})\dfrac{\mathit{decrypt}(n, l, \kappa, \text{d}, [\![t]\!]) = \{et^{\emptyset}\} \qquad \mathit{match}([\![F]\!], et^{\emptyset}) = \sigma \qquad \mathcal{R}' = \mathcal{R}[et \to \emptyset]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{decrypt}(t, F).P]^{\langle n,\kappa \rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup et^{\emptyset} \rangle}}$$

$$(\text{PAR})\dfrac{L \vdash N_1 \rightarrowtail L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail L' \vdash N_1' \parallel N_2}$$

$$(\text{STRUCT})\dfrac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail L' \vdash N'}$$

Figure 2.15: Operational Semantics for acKlaim (unlogged actions), part 1b

in such case the actor (process) will be moved to the other location. Remark that location l will be a **nil** process meaning that the process has terminated in location l, but continuous its execution in l'.

For both rules following conditions must hold for a process to be able to start a new process at a location l':

1. the location l' must exist in the set of nodes Loc and the access policy for l' is δ'. The domain of the location where the new process should run does not matter (as a pc can continue its execution in another pc),
2. the actor must have access to perform the **eval** action at the given location l' and must be able to reach the location in one step.

The reference monitor semantics $\leadsto$ handles these conditions. If these conditions hold, the new process will be added to the resulting net. The new process will have the same key set κ as the actor/process that created it.

The (MOVE) rule describes how processes can move between two locations in one step. We have two rules that describes the effects of the move action.

($\text{MOVE}_{\mathbf{eq}}$) describes the rule when the actor moves away from the location where he started a process into a new location l' that is in the same domain as the ELoc(n). In this case the ELoc(n) will be set to **nil** indicating that the actor has terminated the process and moved to location l'.
The other rule ($\text{MOVE}_{\mathbf{diff}}$) describes the situation when the actor has spawned a process and the process is moving into l' which is in a different domain than ELoc(n). In this case the process will continue its execution in the new location and the actor will still be able to reach the neighbouring locations from ELoc.

The reference monitor is defined such the **move** actions can be performed on the current location of the actor or in an adjacent location. If the ELoc is set then the actor is also able to move to ELoc or in a neighbouring location of ELoc. For an actor n to be able to move to location l' the following conditions must hold:

1. the location l' must exist in the set of nodes Loc and the access policy for l' is δ'.
2. the actor must have access to move to l' and be able to do so in one step either from his current location or from the ELoc.

The reference monitor rule $\rightsquigarrow$ will take care of these two conditions. If all these conditions are hold, the $\mathcal{G}$ component of $\mathcal{S}$, which is the mapping of actors to locations, is updated to reflect that n is moved to l'. In the resulting net an **nil** process is left at location l to ensure that the location does not disappear from the net, as the process that moved could have been the only element at the location. The process that moved will then continue evaluation at l'.

The (ENCRYPT) action takes three arguments, a template field t, an access policy ρ, and a formal field F. The action will try to put the access policy ρ on t and return the result to F. The (ENCRYPT) rule uses the *encrypt* function from the reference monitor semantics to perform the encryption, and if it is successful a substitution is created and applied to the remaining process. The $\mathcal{R}$ part of $\mathcal{S}$, which is the mapping of data to its access policy, is updated to reflect the new access policy ρ and the encrypted datum is added to the actor's key set. The reference monitor semantics will ensure that only data which the actor has access to, can be encrypted, so the actor cannot access data that is not intended for him. The actor will neither be able to encrypt already encrypted data, the policy on the data will be substituted for a new policy if the actor has the proper access.

The (DECRYPT) rule describes the semantics for the **decrypt** action. An actor performing the **decrypt** action will only succeed if he has the necessary access to the datum. Decryption of data will remove all access restriction on the data item and will be added to the actor's key set and become usable as a key. If the decrypt function from the reference monitor semantics is successful, it will return the datum without any security policy. A substitution will be generated that substitutes all references to the decrypted data with its value in the resulting process. The $\mathcal{R}$ component of $\mathcal{S}$ is updated in such that the policy for et is the empty set.

The (PAR) rule states that if a part of a net is reduced, the whole net is reduced accordingly. This makes the rules for actions able to "focus" on small parts of the net. The last rule (STRUCT) relates the structural congruence and the reduction relation by saying that all structural congruent nets can make the same reduction steps.

The semantics for the reduction relations for the logged actions are similar to the semantics in Figure 2.14 and 2.15. The main addition is a global timer T and the global logging component Log. Apart from the these changes the semantic rules are exactly the same as their non-logging counterparts. The rules for the logged actions are found in Figure 2.17.

The last part of the semantics is the rules for template matching. The matching function is used for selecting evaluated tuples from the tuple space according to evaluated templates. It is defined by the rules in Figure 2.16. The match function takes two arguments and returns a substitution σ. The substitution formalizes how a value should be substituted in a process term. For example, P$[l'/u]$ expresses that all occurrences of u should be substituted out for l' in the process term P.

$$\mathrm{match}(\mathrm{V},\,\mathrm{V}) = \epsilon \quad \mathrm{match}(!x,\,\mathrm{V}) = [\mathrm{V}/x] \quad \mathrm{match}(l, l) = \epsilon \quad \mathrm{match}(!u, l') = [l'/!u]$$

$$\frac{\mathrm{match}(\mathrm{F}, ef) = \sigma_1 \ \mathrm{match}(\mathrm{T}, et) = \sigma_2}{\mathrm{match}((\mathrm{F},\,\mathrm{T}),(ef, et)) = \sigma_1 \circ \sigma_2}$$

Figure 2.16: Semantics for template matching

$$(\overline{IN})\;\dfrac{\begin{array}{c} match(\llbracket T\rrbracket, et^{\rho}) = \sigma \qquad \mathcal{K}' = \mathcal{K}[n \rightarrow \mathcal{K}[n] \cup \{et^{\rho}\}] \qquad \mathrm{T} < \mathrm{T}' \\ \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \bar{\mathrm{i}}\rangle} \qquad \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \bar{\mathrm{i}}, l',), l, l', \mathrm{i})] \end{array}}{\begin{array}{c} \mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{in}(T)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle \rightarrowtail \\ \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup \{et^{\rho}\}\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle \end{array}}$$

$$(\overline{OUT})\;\dfrac{\begin{array}{c} l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \llbracket t\rrbracket = et^{\rho} \qquad \mathcal{K}' = \mathcal{K}[l' \rightarrow \mathcal{K}[l'] \cup \{et^{\rho}\}] \\ \mathrm{T} < \mathrm{T}' \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \bar{\mathrm{o}}\rangle} \qquad \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \bar{\mathrm{o}}, l',), l, l', \mathrm{o})] \end{array}}{\mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{out}(t)@l'.P]^{\langle n,\kappa\rangle} \rightarrowtail \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} \langle et^{\rho}\rangle}$$

$$(\overline{EVAL}_{\mathbf{nil}})\;\dfrac{\begin{array}{c} l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \mathrm{ELoc}' = \mathrm{ELoc}[n \rightarrow l] \qquad \mathrm{T} < \mathrm{T}' \\ \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \bar{\mathrm{e}}\rangle} \qquad \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \bar{\mathrm{e}}, l'), l, l', \mathrm{e})] \end{array}}{\begin{array}{c} \mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(Q)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \rightarrowtail \\ \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \end{array}}\;\mathrm{ELoc}[n] = \mathbf{nil}$$

$$(\overline{EVAL}_{\mathbf{def}})\;\dfrac{\begin{array}{c} l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \bar{\mathrm{e}}\rangle} \\ \mathrm{T} < \mathrm{T}' \quad \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \bar{\mathrm{e}}, l'), l, l', \mathrm{e})] \end{array}}{\begin{array}{c} \mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(Q)@l'.P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \rightarrowtail \\ \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \end{array}}\;\mathrm{ELoc}[n] = \mathbf{nil}$$

$$(\overline{MOVE}_{\mathbf{diff}})\;\dfrac{\begin{array}{c} l' \in \mathrm{Loc} \qquad \mathcal{R}(l') = \delta' \qquad \mathcal{G}' = \mathcal{G}[n \rightarrow l'] \\ \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \overline{\mathrm{m}}\rangle} \qquad \mathrm{T} < \mathrm{T}' \\ \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \overline{\mathrm{m}}, l'), l, l', \mathrm{m})] \end{array}}{\begin{array}{c} \mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa\rangle} \rightarrowtail \\ \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \end{array}}\;\mathrm{Dom}(l) \neq \mathrm{Dom}(\mathrm{ELoc}(n))$$

$$(\overline{MOVE}_{\mathbf{eq}})\;\dfrac{\begin{array}{c} l' \in \mathrm{Loc} \quad \mathcal{R}(l') = \delta' \quad \mathcal{G}' = \mathcal{G}[n \rightarrow l'] \quad \mathrm{T} < \mathrm{T}' \\ \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \overline{\mathrm{m}}\rangle} \qquad \mathrm{ELoc}' = \mathrm{ELoc}[n \rightarrow \mathbf{nil}] \\ \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (grantby(n, l, \kappa, \overline{\mathrm{m}}, l'), l, l', \mathrm{m})] \end{array}}{\begin{array}{c} \mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa\rangle} \rightarrowtail \\ \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \parallel l' ::^{\delta'} [P]^{\langle n,\kappa\rangle} \end{array}}\;\mathrm{Dom}(l) = \mathrm{Dom}(\mathrm{ELoc}(n))$$

$$(\overline{DECRYPT})\;\dfrac{\begin{array}{c} decrypt(n, l, \kappa, \mathrm{d}, \llbracket t\rrbracket) = \{et^{\emptyset}\} \qquad match(\llbracket T\rrbracket, et^{\emptyset}) = \sigma \qquad \mathcal{R}' = \mathcal{R}[et \rightarrow \emptyset] \\ \mathrm{T} < \mathrm{T}' \qquad \mathrm{Log}' = \mathrm{Log}[\mathrm{T} \mapsto (decryptby(n, l, \kappa, \bar{\mathrm{d}}, \llbracket t\rrbracket), l, l', d)] \end{array}}{\mathrm{Log}, \mathrm{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{decrypt}(t, T).P]^{\langle n,\kappa\rangle} \rightarrowtail \mathrm{Log}', \mathrm{T}', \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup et^{\emptyset}\rangle}}$$

Figure 2.17: Operational Semantics for acKlaim (logged actions), part 2

2.11 Analysis

This section gives an overview of the two analysis presented in the work. Both of these analyses are based on static program analysis techniques which is the discipline of extracting and deriving information that holds for every single configuration of the analysed system. Being based on a formal system model, from an initial configuration all possible states of the system are computed.

The first analysis is called *Conditional Reachability Analysis* and determines which locations in a system an actor with name n and keys κ can reach from location l. The analysis makes a map over which locations and data an insider can reach. The results of this analysis would be useful for a "before-the-fact" analysis, to identify vulnerabilities in the system.
The second analysis is called *Log-trace Reachability Analysis* which is an after-the-fact analysis that analyses the system model supplied with a log file to explore what an actor might have done in between two log entries. The analysis makes an estimate for which locations were reached and which data was accessed by an actor, by simulating the sequence of actions provided in the log file.

2.11.1 Log-Equivalency

The papers on the ExASyM model present two algorithms in which both of them use the notation of "log-equivalency". Log-equivalency is an analysis that finds locations and actions that from the viewpoint of an observer cannot be distinguished. The analysis is used to find out that an actor can be in a location l, then he might just as well be in an any equivalent location or might have performed any actions between. Therefore two locations and/or actions are indistinguishable from the view of the analysis, if the actor does not cause a log entry. Log-equivalency is needed to find out what might have happened between two log entries.

The pseudo-code in Algorithm 4 shows the realisation of log-equivalency. The main idea of the algorithm is to visit all locations where a user might be and compute the effect of every unlogged action that the user is allowed to perform from that location. The computation is repeated until no further changes to the graph occurs.

Algorithm 4 equivalent

```
procedure EQUIVALENT()                                ▷ perform (log-)equivalent actions
    *changed* = true
    while *changed* do
        *changed* = false
        for all actors $n$ do
            for all locations $l$ that $n$ might be located at do
                for all locations $l'$ reachable from $l$ in one step do
                    simulate all actions that $n$ can perform on $l'$ (without caus-
                    ing a log entry in case of LTRA)
                    for each action set changed if $n$ at location $l$ learns a new
                    data item
                end for
            end for
        end for
    end while
end procedure
```

2.11.2 Conditional Reachability Analysis

The first analysis is a reachability analysis where we are interested in a subset of data and locations a given actor can reach. The result of the analysis can be used in computing which data an actor with name n and keys κ can reach from location l, by evaluating actions he can execute from the locations he can reach.

When designing a system, especially an access-control system, it rapidly becomes unclear, which parts of the system are accessible by which users. In systems combining networks with real buildings, the distinction between reachable and unreachable becomes even more blurry. In this scenario, the first analysis may be applied in order to gain a better understanding of capabilities of actors by identifying which places a user may reach, based on who he is, what he knows, and where he is located.
Hence the name "before-the-fact" used to ensure whether a given system lives up to a set of access-control restrictions we would like to know before an attack occurs. The analysis can be served for the purpose of tightening the access control system to hinder actors from reaching certain areas of the modelled system.

The algorithm for CRA is given in Algorithm 5. Essentially it only sets up the analysis by initialising all data structures, followed by a single call to `equivalent`, which performs the simulation unconstrained for all possible actions and iterates until no further changes occur. In order to avoid non-termination it keeps track of which actor with which knowledge has been at which location, thus re-analysing already seen scenarios can be avoided.

Algorithm 5 Conditional Reachability

```
1: procedure ConditionalReachability (system specification)
2:     /* initialization */
3:     place all actors at their initial location
4:     initialize all actors and location with initial key set
5:     /* computer all actions that can be performed */
6:     eqiuvalent()
7: end procedure
```

2.11.3 Log-trace Reachability Analysis

The second analysis takes into account what the actor actually does in the system. The analysis is based on a log file together with a system model as described in previous sections. The objectives of this analysis is to be able to identify, what has happened in the system before, under, and after an attack has happened.
It should be remarked that the amount of data logged may be limited due to privacy concerns for our modern society, thus also limiting how useful the logged information is. Therefore it is unrealistic to assume that all actions in the system are logged, and hence the analysis is based on a log file with incomplete information about actions performed in the system. The analysis will have to take into account the actions that are logged, and simulate all possible actions that might have happened (unnoticed) in between two log entries for each and every actor in the system. The more fine-grained the logging system is, the more precise the result of this analysis will be, but the more coarse-grained the logging system is, it becomes harder to keep a clear view of what might actually have happened in between.

The analysis traces for all users all possible ways that they might take through the system. However, in this analysis the set of all paths is restricted by the requirement that the actions performed must match the logged events. This restriction results exactly in what is needed for the "after-the-fact" analysis, a tool that matches logged actions against possible actions, thus identifying locations that an actor might have reached and data items an actor might have accessed unnoticed.

The log-trace reachability analysis is not used in this project, hence the algorithm is omitted. For interested readers, please consult [PH08] in which the pseudo-code of the algorithm is explained in greater details. Basically it is an graph-based algorithm that evaluates, what effect sequences of logged actions might have had, by evaluating all sequences of actions on the system representation.

Chapter 3

Model Checking

Our reliance on the functioning of IT-systems is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life. Besides of offering good performance in terms like response times and processing capacity, the absence of annoying errors is one of the major quality indications.
Modern IT-systems are no longer standalone applications, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They thus become much more vulnerable to errors – the number of defects grows exponentially with the number of interacting system components [BK08].

Model checking is a verification tool that is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary e.g. a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the system's specification. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfil one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness in this sense is always relative to a specification, and is not an absolute property of a system.

Model checking is a technique that are based on models describing the possible system behaviour in a mathematically precise and unambiguous manner. Using this model which explores all possible system states in a systematic manner. In this way, it can be shown that a given system model truly satisfy a certain property specification.
The property specification prescribes what the system should do, whereas the model description addresses how the system behaves. The model checker provides a counterexample that indicates how the model could reach an undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information and adapt the model (or the property) accordingly.

In order to wrap up the discussion one can describe model checking in a nutshell as an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for a given state in that model.

3.1 Phases in Model Checking

There are basically three phases in model checking, *modelling*, *running* and *analysis* phases. I will briefly go through each phases in this section.

The prerequisite inputs to model checking are a model of the system under consideration and a formal characterization of the property to be checked.
Models of systems describe the behaviour of system in an accurate and unambiguous way. They are mostly expressed using finite-state automata, consisting of a finite set of states and transitions. States comprise information about the current values and variables of the system. Transitions describe how the system evolves from one state into another.
To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language (temporal logic). This language allows for the specification of a broad range of relevant system properties such as functional correctness ("*does the system do what it is supposed to do*"), reachability ("*is it possible to end up in a critically vulnerable state?*"), safety ("*something bad never happens*"), liveness ("*something good will eventually happen*") and fairness ("*does under certain conditions, and event occur repeatedly?*") properties.

When the model and the properties has been specified the model checker is ready to carry out the exhaustive verification. This is basically an algorithmic approach in which the validity of the property under consideration is checked in all states of the system model.

There are basically three possible outcomes of a model-checker: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.
In the case the property is valid the following property can be checked or in case all properties have been checked, the model is concluded to possess all desired properties.
Whenever a property is falsified it may have different causes. There may be a modelling error (e.g. the model does not reflect the design of the system). This implies a correction of the model and verification has to be restarted with the improved model. This reverification includes the verification of those properties that were checked before on the erroneous model and whose verification may be invalidated by the model correction. If the error analysis shows that there is no undue discrepancy between the design and its model, then either a design error has been exposed, or a property error has taken place. In case of design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out. As the model is not changed, no reverification of properties that were checked before has to take place. The design is verified if and only if all properties have been checked with respect to a valid model.

A way to deal with models that is too large to be handled in the physical memory of the computer is to make a representation of state spaces using symbolic techniques such as binary decision diagrams or partial order reduction. Another way of dealing with state spaces that are too large is to give up the precision of the verification result.

The list below recaps the important founding for each phase.

Modelling phase

- Model the system under consideration using the model description language of the model checker (e.g. in UPPAAL or PRISM).
- Formalize the property to be checked using the property specification language.

Running phase

- Run the model checker to check the validity of the property in the system model.

Analysis phase

- If the property satisfied, then check next property (if any).
- If the property violated, then:
  - Analyze generated counterexample by simulation.
  - Refine the model, design or property.
  - Repeat the entire procedure.
- If the model checker is out of memory then try to reduce the model and try again.

3.1.1 Counterexample-Guided Abstraction Refinement

The main challenge in model checking is the state explosion problem that can occur if the transition system being verified is very large. For solving this issue a number of state reduction approaches have been proposed to reduce the number of states in the model. Some state reduction techniques exists following: symmetry reductions, partial order reductions and abstraction techniques. Among these techniques, abstraction is considered as the most general and flexible for handling the state explosion problem [CGJ$^+$03].
Intuitively, abstraction amounts to removing or simplifying details as well as removing components of the original design that are irrelevant to prove or disprove the system's correct behaviour. This has the advantage of verifying the simplified model should be more efficient than verifying the original model.

The information loss incurred by simplifying the model however has a price; verifying an abstract model potentially leads to wrong results, at least if the abstraction performed naively.
Hence, the abstraction techniques can be distinguished depending on how the information loss is performed. Over-approximation and under-approximation are two techniques used keep the error one-sided. That is, they suffer for finding false negatives and false positives, respectively. This is shown in Figure 3.1, in which the figure in the middle can obtain results in the over-approximated universe which is not a correct answer in the original design. The figure on the right shows the problem that can happen for under-approximation. Finding the optimum (coarsest) abstraction in polynomial-time is said to be NP-hard. This is shown in the leftmost figure.

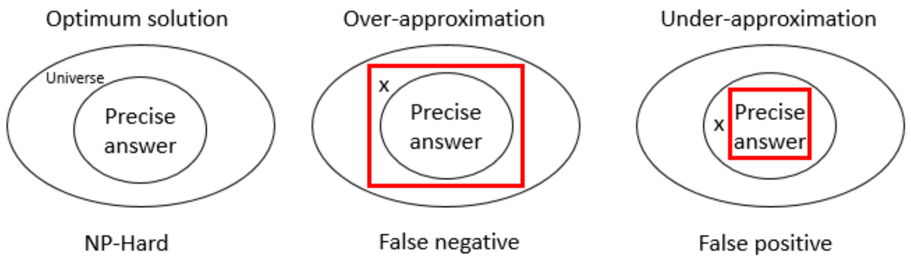


Figure 3.1: Over- and under-approximation

Over-approximation techniques systematically release constraints, thus enriching the behaviour of the system. They establish a relationship between the abstract model and the original one such that correctness at the abstract level implies correctness of the original system. In contrast, under-approximation techniques systematically remove irrelevant behaviour from the system so that a specification violation at the abstract level implies a specification violation of the original system.

In practice, abstraction based methods have been essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, which requires considerable creativity and insight. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions.

Counterexample-guided abstraction refinement (CEGAR) is a automated technique that iteratively refines an abstract model using counterexamples. A counterexample is a witness of a property violation and are given as “error paths”, i.e., paths through the transition diagram that violates a given property.

CEGAR starts with a relatively small skeletal representation of the system to be verified, increasingly compute a more refined and precise abstract representations of the system. The key step in CEGAR is to extract information from false negatives due to over-approximation. This initial abstract model can be obtained by partitioning the state space of the transition system under consideration into clusters of states, and treating the clusters as new abstract states. Then the model checker analyses the abstracted model together with a property specification. If it does not find an error path, meaning that a specification in the temporal logic is true, then it will also be true in the concrete design. The analysis terminates, reporting that no violation exists.
However, if the specification is false in the abstract model, the counterexample may be the result of some behaviour in the approximation that is not present in

the original model, then the counterexample (error path) is checked for feasibility, i.e., if the path is executable according to the concrete system. If the error path is feasible, the analysis terminates, reporting the violation of the property, together with the feasible error path as witness. On the other hand, if the error path is infeasible, the violation is due to a too coarse abstract model and the infeasible error path is used to automatically refine the current abstraction.
The refinement it done so the behaviour that caused the erroneous counterexample is eliminated and the process of reverification of the property specification restarts. This gives rise to a sequence of increasingly precise abstract models till the specification is either proved or disproved by a counterexample.
A way to do this refinement is to identify the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix (the “failure state”) needs to be split into less abstract states by refining the equivalence classes in such a way that the “false” counterexample is eliminated. Thus, a more refined abstraction function is obtained. Note that there may be many ways of splitting the failure state; each determines a different refinement of the abstraction. It is desirable to obtain the coarsest refinement which eliminates all the counterexample because this corresponds to the smallest abstract model that is suitable for verification however, finding the coarsest refinement is NP-hard.

A generic algorithm of the CEGAR is shown in Algorithm 6.

Algorithm 6 CEGAR

```
function CEGAR($M, \varphi$)
    /* The coarsest abstraction that can be found using
    construction heuristic */
    $M'$ = Initial abstract model($M$)
    while $true$ do
        if $M' \models \varphi$ then
            return $safe$                                ▷ the abstraction is safe
        else
            $\sigma$ = extractCounterExample($M', \varphi$)
            if isFeasible($\sigma, M$) then
                return $unsafe$                  ▷ error path is feasible, report bug
            else
                /* refine the abstraction such that the cur-
                rent flaw doesn't appear in next iteration */
                refine($M'$)
            end if
        end if
    end while
end function
```

3.2 Transition Systems

Transition systems are often used in computer science as models to describe the behaviour of systems. They are basically directed graphs where nodes represent states and edges model transitions. i.e. state changes. A state describes some information about a system at a certain moment of its behaviour. Depending on what kind of system is modelled, this can almost be anything, e.g. a state of a thermometer in a freezer indicates the current temperature of the freezer. Transitions specify how the system can evolve from one state to another. In model checking it is common to use transition systems with action names for the transitions and atomic propositions for the states.

Definition 7: Transition System (TS)

A transition system TS is a tuple $TS = (S, Act, \longrightarrow, I, AP, L)$ where

- S is a set of states.
- Act is a set of actions.
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation (in which the first and third element in the triplet is the source and target state respectively and the element in the middle is the action of the transition).
- $I \subseteq S$ is a set of initial states.
- AP is a set of atomic propositions.
- $L : S \to 2^{AP}$ is a labelling function (2^{AP} is the power set of AP).

The TS is called *finite* if S, Act and AP are finite. $\square$

The behaviour of a transitions system can be described as follows. The transitions system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\longrightarrow$. That is of s is the current state, then a transition $s \xrightarrow{\alpha} s'$ originating from s is selected non-deterministically and taken, i.e. an action α is performed and the transition system evolves from state s into state s'. This selection procedure is repeated in state s' and finished once a state is encountered that has no outgoing transitions. Such a state is called terminal if and only if it has no successor state. For transition systems where terminal states occurs as a natural phenomenon represents the termination of the program.
It is important to realize that in case a state has more than one outgoing transition, the "next" transition is chosen in purely non-deterministic fashion. That

is, the outcome of this selection process is not known a priori, and hence no statement can be made about the likelihood with which a certain transition is selected. Similarly, when the set of initial states consists of more than one state, the start state is selected non-deterministically.
The transitions in the TS corresponds to the advance of a single time-unit. The underlying time domain is thus discrete. The present moment refers to the current state and the next moment corresponds to the immediate successor state.

Modelling unreliable and unpredictable system behaviour can be to some extend be modelled by non-determinism. This is often appropriate in early system design phases where systems are considered at a high level of abstraction and where information about the likelihood is (sometimes deliberately) left unspecified. In later design stages, though, where the internal system characteristics become more dominant, probabilities are a useful to quantify and thus refine this information. This could be especially by useful in terms of insider attack where one would like to derive information about the maximum probability for reaching a critically vulnerable state of an organization.
In order to model random phenomena, transition systems are enriched with probabilities. This can be done in different ways. In discrete-time Markov chains (MC), all choices are probabilistic. Markov chains are the most popular operational model for the evaluation of performance and dependability of information-processing systems [BK08].

Roughly speaking, Markov chains behave as transition systems with the only difference that non-deterministic choices among successor states are replaced by probabilistic ones. That is to say, the successor state of state s is chosen according to a probability distribution. This probability distribution only depends on the current state s, and not on, e.g., the path fragment that led to state s from some initial state. Accordingly, the system evolution does not depend on the history, but only on the current state s. This is known as the *memoryless* property.

Definition 8: (Discrete-Time) Markov Chain (MC)

A (discrete-time) Markov chain is a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{init}, AP, L)$ where

- S is a finite, non-empty set of states.
- $\mathbf{P} : S \times S \to [0, 1]$ is the transition probability function such that for all states s:
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1$$

- $\iota_{init} : S \to [0,1]$ is the initial distribution such that $\sum\limits_{s \in S} \iota_{init}(s) = 1$

- AP is a set of atomic propositions.

- $L : S \to 2^{AP}$ is a labelling function.

The $\mathcal{M}$ is called *finite* if S and AP are finite. □

The transition probability function $\mathbf{P}$ specifies for each state s the probability $\mathbf{P}(s, s')$ of moving from s to s' in one step, i.e., by a single transition. The constraint imposed on $\mathbf{P}$ ensures that $\mathbf{P}$ is a distribution.

The value $\iota_{init}(s)$ specifies the probability that the system evolution starts in state s. The states s with $\iota_{init}(s) > 0$ are considered as the initial states. In a similar way, the states s' for which $\mathbf{P}(s, s') > 0$ are viewed as the possible successors of s.

It is a common to identify the transition probability function $\mathbf{P} : S \times S \to [0,1]$ with the matrix $(\mathbf{P}(s,t))_{s,t \in S}$. The row $\mathbf{P}(s, \cdot)$ for state s in this matrix contains the probabilities of moving from s to its successors, while the column $\mathbf{P}(\cdot, s)$ for state s specifies the probabilities of entering state s from its predecessor state. Similarly, the initial distribution ι_{init} is often viewed as a vector $(\iota_{init}(s))_{s \in S}$.

The use of atomic propositions and the labelling function L is the same as for transition systems mentioned above.

Markov chains are depicted by their underlying directed-graph where edges are equipped with the transition probabilities in]0,1]. If a state s has a unique successor s', i.e., $\mathbf{P}(s, s') = 1$, the transition probability may sometimes be omitted.

3.2.1 Probabilities of Markov Chains

One of the main advantages of Markov chains is that it allows to easily calculate the probability of being in any state after short- and long-term behaviour of the process in a convenient way. In this section we are looking at how to calculate some of the probabilities.

3.2.1.1 Transient State Probabilities

The transient state describes the short-run behaviour of the Markov chain. It basically computes the probability of being in a state after n steps.

Let $\mathbf{P}$ be the probability transition matrix and let ι_{init} be the probability vector which represents the starting distribution of a Markov chain in consideration. Then the probability that the chain is in state s_i after n steps is the ith entry in the vector $\mathbf{v}$.

$$\iota_{init} \cdot \mathbf{P}^n = \mathbf{v} \tag{3.1}$$

The nth power of $\mathbf{P}$, i.e., the matrix $\mathbf{P}^n$, contains the state probabilities after exactly n steps (i.e., transitions) inside the state space S of the Markov chain. More precisely, let $s \in \iota_{init}$ be an initial state, and $s' \in Post(s)$ be a successor state of s, then the matrix entry $\mathbf{P}^n(s', t)$ multiplied with $\iota_{init}(s)$ equals the sum of the probabilities $\mathbf{P}(s_0, s_1, \ldots, s_n)$ of all path fragments $s_0, s_1, \ldots, s_n$ with $s_0 = s, s_1 = s', s_n = t$ and $s_i \in S$ for $0 \leq i \leq n$.

$Post(s)$ defines the successor state and is formally defined as:

$$Post(s) = \{s' \in S \mid \mathbf{P}(s, s') > 0\}$$

3.2.1.2 Steady-State Probabilities

Steady-state probabilities are used to describe the long-run behaviour of a Markov chain. The steady-state probabilities implies that the transition probabilities stops changing in the long run. Every Markov chain have a unique steady-state distribution, if the Markov chain is regular. A Markov chain is regular if its transition matrix $\mathbf{P}$ is regular. That is if some power of $\mathbf{P}$ has only positive entries. With other words steady-state probabilities exists for aperiodic, and strongly connected Markov chain. It does neither depend on the initial distribution [Her13].

A distribution $\mathbf{v}$ is stationary, if:

$$\mathbf{v} \cdot \mathbf{P} = \mathbf{v} \tag{3.2}$$

It should also be remembered that the sum of the vector $\mathbf{v}$ has to add up to 1. This is an important feature in order to find the steady-state probabilities.

$$\sum_{i=0}^{n-1} \mathbf{v}(i) = 1$$

3.2.2 Example

For this example we consider the running example presented in section 2.3. The transition system shown in Figure 3.2 depict the locations the janitor can reach from his initial location (janitor's workshop). From the workshop he can of course stay at his workshop and do some stuff or go to the hallway with probability of 0.3. From the hallway he can further move to other locations. The janitor can't access the user's office as he don't have the cipher-key for the room, therefore it is not shown on the graph. For each transition the probability that the janitor will move to that location is depicted of the transition diagram. The values used for this example is purely fictional.

The transition probability function $\mathbf{P}$ can viewed as a 9×9 matrix (as there are 9 states in the transition diagram) and the initial distribution viewed as a vector ι_{init}.

$$\mathrm{M} = \begin{array}{c|ccccccccc|} & Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ Jan & .7 & .3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ Hall & 0 & .1 & .4 & 0 & 0 & .2 & 0 & 0 & .3 \\ Bsmt & 0 & .15 & .15 & .3 & .4 & 0 & 0 & 0 & 0 \\ waste & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ vault & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ FR & 0 & .5 & 0 & 0 & 0 & 0 & .5 & 0 & 0 \\ Rec & 0 & 0 & 0 & 0 & 0 & .6 & .25 & .15 & 0 \\ Out & 0 & 0 & 0 & 0 & 0 & 0 & .65 & .35 & 0 \\ L_{jan} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\iota_{init} = \begin{array}{ccccccccc} Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ (1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) \end{array}$$

Since $\mathbf{P}$ is a stochastic matrix, i.e., each row sum equals 1, s is an absorbing state if and only if $\mathbf{P}(s, s) = 1$, and $\mathbf{P}(s, t) = 0$ for all states $t \neq s$.

With the information available in the matrix we can calculate the probability distribution of where the janitor might be after 5 steps, using the formula in 3.1.

$$\iota_{init} \cdot \mathbf{P}^5 = \mathbf{v}$$

The detailed calculations can be found in Appendix A.1. The results are shown

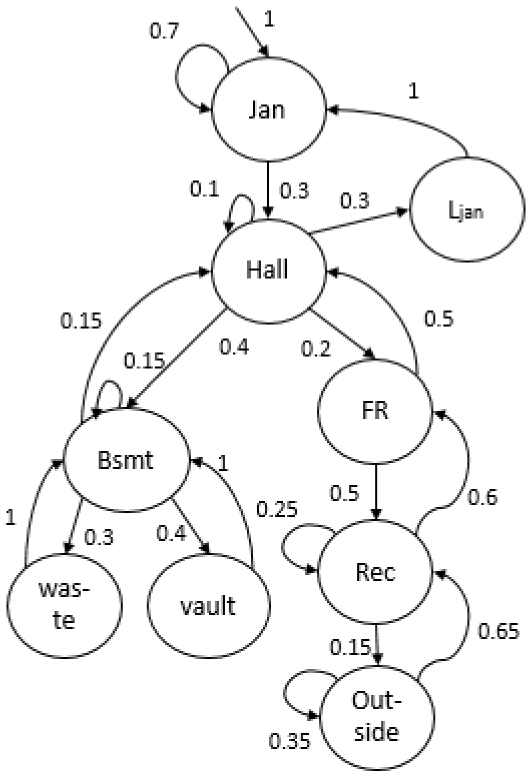


Figure 3.2: Markov Chain for the running example (janitor)

below:

$$\begin{array}{cccccccccc} & Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ \mathbf{v} = & \big(.328 & .191 & .185 & .056 & .075 & .057 & .041 & .006 & .057 \big) \end{array}$$

From this we can conclude that after 5 steps within the system, the probability that the janitor will be in his workshop is 32.8%.
Another interesting property we can se from the results is that if we add up the probability for being in the basement together with the waste bin and the vault where important documents are stored we obtain the value 31.6%. This means that there is almost even probability that the janitor will be at the basement or at his workshop after 5 steps.

Using the equation 3.2 we can obtain the steady-state probability distribution.

This will tells us the probability of where the janitor might be located after long-term behaviour of the transition system. Again the detailed calculations can be found in the appendix A.2. The results is shown in the vector below:

$$\begin{array}{cccccccccc} & Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ \mathbf{v} = & (.130 & .130 & .348 & .104 & .139 & .052 & .043 & .010 & .039) \end{array}$$

From these results we can conclude that the likelihood that the janitor will be at his workshop in the long-run is only 13% regarding to the 32.8% from the previous results. Also an interesting observation that can be made here is that the probability that the janitor is located at the basement is 34.8%. This high probability could signal that the janitor could obtain some sensitive data from the vault in the long run. In terms of the insider attack this could mean that this room should be put under special scrutiny or perhaps that the lock to the vault should be taken from the janitor, and only be given to him when he really needs it.

3.3 Temporal Logic

Probabilistic computation tree logic (PCTL, for short) is used for expressing system properties for Markov chains. It is a branching-time temporal logic, based on the logic CTL. CTL (used for TS) is also a branching temporal logic that is based of propositional logic with a discrete notion of time. Branching time refers to the fact that at each moment there may be several different possible futures. Each moment of time may thus split into several possible futures. Due to this branching notion of time, this class of temporal logic is known as branching temporal logic. The semantics of a branching temporal logic is defined in terms of an infinite directed tree of states. Each traversal of the tree starting in its root represents a single path. The tree itself thus represents all possible paths and is directly obtained from a transition system by "unfolding" at the state of interest (shown in Figure 3.3).

The verification of probabilistic systems can be focused on either quantitative properties or qualitative properties (or both). Quantitative properties typically put constraints on the probability or expectation of certain events. Instances of quantitative properties allows to ask questions like *"eventually the janitor will get the secret from the vault with probability at most $\frac{1}{4}$"*.

$$\mathbb{P}_{\leq 0.25}(\Diamond in\_J\_secret)$$

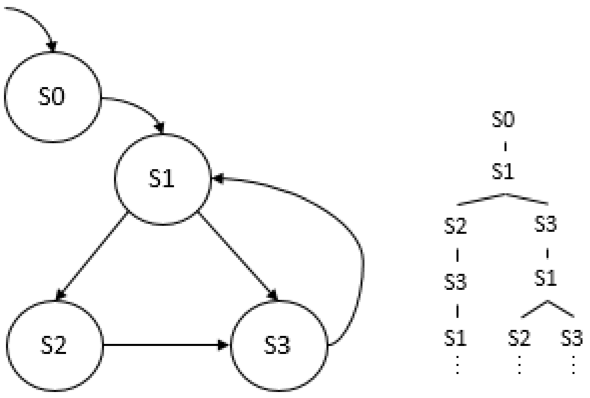


Figure 3.3: A transition system and a prefix of its infinite computation tree

Or alternatively assert that the probability that the receptionist will enter the hallway after the user went home within 5 steps (time units) is at least 0.15.

$$\mathbb{P}_{\geq 0.15}(move\_U\_Out \cup^{\leq 5} move\_R\_Hall)$$

Qualitative properties on the other hand, typically assert that a certain (good) event will happen almost surely, i.e., with probability 1 or dually, that a certain (bad) event almost never occurs, i.e., with zero probability. Typical qualitative properties for Markov chains could be questions of reachability (does eventually an event always hold?).

A PCTL formula formulates conditions on a state of a Markov chain. The interpretation is Boolean, i.e., a state either satisfies or violates a PCTL formula. The logic PCTL is defined like CTL, PCTL incorporates, the probabilistic operator $\mathbb{P}_J(\varphi)$ where φ is a path formula and J is an interval of [0, 1]. The path formula φ imposes a condition on the set of paths, whereas J indicates a lower bound and/or upper bound on the probability. The intuitive meaning of the formula $\mathbb{P}_J(\varphi)$ in state s is: the probability for the set of paths satisfying φ and starting in s meets the bounds given by J.

Definition 9: Syntax of PCTL

PCTL state formulae over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi ::= \texttt{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\varphi)$$

where $a \in AP, \varphi$ is a path formula and $J \subseteq [0,1]$ is an interval with rational bounds. PCTL path formulae are formed according to the following grammar:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \cup \Phi_2 \mid \Phi_1 \cup^{\leq n} \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae and $n \in \mathbb{N}$. □

The propositional logic fragment of PCTL, as well as the path formulae $\bigcirc\Phi$ and $\Phi_1 \cup \Phi_2$ has the same meaning as in CTL. The formula $\bigcirc\Phi$ holds for a path if Φ holds at next state in the path and $\Phi_1 \cup \Phi_2$ holds for a path if there is some state along the path for which Φ_2 holds, and Φ_1 holds in all states prior to that state.
Path formula $\Phi_1 \cup^{\leq n} \Phi_2$ is the step-bounded variant of $\Phi_1 \cup \Phi_2$. It asserts that the event specified by Φ_2 will hold within at most n steps, while Φ_1 holds in all states that are visited before a Φ_2-state has been reached. The semantics of this can be more formally described as follows:

$$\begin{array}{lll}
\pi \models \bigcirc\Phi & \text{iff} & \pi[1] \models \Phi \\
\pi \models \Phi \cup \Psi & \text{iff} & \exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j : \pi[k] \models \Phi)) \\
\pi \models \Phi \cup^{\leq n} \Psi & \text{iff} & \exists 0 \leq j \leq n.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j : \pi[k] \models \Phi))
\end{array}$$

where the path $\pi = s_0, s_1, s_2 \ldots$ and integer $i \geq 0, \pi[i]$ denotes the $(i+1)$-st state of π

Other Boolean connectives are derived in the usual way, e.g., $\Phi_1 \vee \Phi_2$ is obtained by $\neg(\neg\Phi_1 \wedge \neg\Phi_2)$ (De Morgan's low). The eventually (future) operator ($\Diamond$) can be derived as follows: $\Diamond\Phi = \mathit{true} \cup \Phi$. Similarly, for step-bounded eventually we have: $\Diamond^{\leq n}\Phi = \mathit{true} \cup^{\leq n} \Phi$. A path satisfies $\Diamond^{\leq n}\Phi$ if it reaches a Φ-state within n steps.
The always (globally) operator $\Box$ can be derived using the duality of eventually and always and the duality of lover and upper bounds. Thus it is possible to define $\mathbb{P}_{\leq p}(\Box\Phi) = \mathbb{P}_{\geq 1-p}(\Diamond\neg\Phi)$ and $\mathbb{P}_{]p,q]}(\Box^{\leq n}\Phi) = \mathbb{P}_{[1-q,1-p[}(\Diamond^{\leq n}\neg\Phi)$.

3.4 Generating LTS

In this section we are looking at the process on how to derive the LTS from the ExASyM model. For this transformation we need a model (data structure) which will be introduced here. This model should allows us to perform probabilistic model checking.

Definition 10: graph-based ExASyM model

The ExASyM model under consideration can be seen as a graph-based model that is specified in the following tuple-form: $\mathcal{M} := (\mathcal{N}, \mathcal{E}, \mathcal{D}, \mathcal{A}, \mathcal{AC})$ where:

$\mathcal{N}$: set of nodes representing locations in an organization.
$\mathcal{E} \subseteq \mathrm{N} \times \mathrm{N}$: is a set of directed edges representing connection between locations.
$\mathcal{D}$: set of data items.
$\mathcal{A}$: set of actors.
$\mathcal{AC} \subseteq (\mathcal{N} \cup \mathcal{D}) \times 2^{(\mathcal{A} \cup \mathcal{D} \cup \mathcal{N} \cup \star)} \times 2^{actions}$: access control restriction for locations and data items. □

The set of actions supported by ExASyM are the standard action that has been presented in section 2.6. The LTS we want to model has the following tuple structure.

Definition 11: LTS specification for ExASyM

The corresponding LTS for a given ExASyM model is a 4-tuple system: $LTS(\mathcal{M}) = (S, A, T, s_0)$ where:

S = Conf: the set of valid configurations of the system, that comply with the access control specification. A configuration is a set of tuples in form of ($\{a$, $\mathcal{D}(a)\}$, l, $\mathcal{D}(l)$), where a $\subseteq \mathcal{A}$, $\mathcal{D}(a)$, $\mathcal{D}(l) \subseteq \mathcal{D}$, $l \in \mathcal{N}$. This tuple represents the set of actor and the data items each actor has located in l, and the data available at location l.

A = Actions: set of action labels that are representing which action is performed, who is the actor performing the action, and to which location. It has the following format: `actionname_action_locationname`.

$T \subseteq$ S × A × S: set of transitions. Each transition has a probability in the rational bounds of the interval [0;1].

$s_0 \in$ S: the initial configuration (state) of the system. □

3.5 Transforming ExASyM to LTS

The first step in the transformation procedure is to perform a reachability analysis in order to generate the set of valid configurations and the transitions. The analysis visits all locations where a user might be, performs all actions that the actor does not need another data item for, and generates states and transitions accordingly. The algorithm also checks if meanwhile a new data item has been obtained or a new location has been reached. If so, perform all the actions that this capability provides the actor. The computation is repeated until no further changes to the graph occur. The main principle algorithm was sketched in Algorithm 4. In this section we will go one step further and dive into the details of the algorithm.
The transformation algorithm consists of three parts. The first part initializes the data structures needed for the algorithm, and creates the initial state of the LTS. Part 2 finds all permissible actions that can be performed from the standpoint of the actor's current location and the data items he holds. The last part simulates each individual action and their side effects on the LTS and the system.

Part 1 initializes the data structures for needed for the LTS. It is assumed that these data structures are public and available for the other parts of the algorithm as well. After the initialization, the initial state is set-up. Remember that a state consist of a set of valid configurations and a configuration consists of a location, the set of data items and actors in that location. Each actor also have their set of data items. When this part is over a call to `simulateActions` is made (part 2 of the algorithm), which will find and perform all permissible actions for each user from the initial state and extend the LTS until a fixed point is reached. Algorithm 7 sketches the pseudo-code of the first part of the algorithm.

The second part of the algorithm (Algorithm 8) tries to simulate all permissible actions that an actor can perform from his current location and from the ELoc (if it is set). Notice that the last part checks access from the ELoc to its neighbouring location, l', and if there is a permissible action that can be executed, it will be performed from the current location l to the reachable location l'. This means that if the actor is currently in a location with a different domain where he originally started from, he will also be able to reach every location from the ELoc. Intuitively this could indicate that the actor left the other domain (e.g. logged off from the computer).

Algorithm 7 LTS(ExASyM)

```
function LTS($ExASyM$)
    /* initialize the state space, set of actions and
    transitions */
    $S = \{\}$
    $A = \{$m, mlog, e, elog, i, ilog, o, olog$\}$
    $T = \{\}$
    $s0 = \{\}$

    /* configure initial state */
    for all $l \in$ Loc do
        $conf = (\{\}, l, \mathcal{D}(l))$
        for all $n \in$ Actor do
            for all $l' \in$ Loc$(n)$ do
                if $l = l'$ then
                    $conf$.add$(n, \mathcal{D}(n))$
                end if
            end for
        end for
        $s0 = s0 \cup conf$
    end for
    $S = S \cup s0$
    simulateActions($ExASyM$)
    $LTS = (S, A, T, s0)$
     return $LTS$
end function
```

The call to the `simulateAction` function will return the state that reflects the outcome of the action performed. When all states has been handled, such that a fixed point is reached the algorithm will stop.

The last part of the algorithm is quite detailed (Algorithm 9). This part will handle each action separately depending on the action. If the outcome of the action results in a new state that has not been discovered until now, then this state will be returned, otherwise a **nil** state is returned.

A helper function has been introduced in order to decrypt keys the actor holds when he moves into a new location. The function is called `decryptKeys` and will decrypt all keys in the key set that is possible. The pseudo-code is shown in Algorithm 11.

Algorithm 8 simulateActions(ExASyM)

```
function SIMULATEACTIONS($ExASyM$)
    $states = S$
    repeat
        $s = head(states)$
        $states = states \backslash s$
        for all $(\{(n, \mathcal{D}(n))\}, l, \mathcal{D}(l)) \in s$ do
            for all $(n, \mathcal{D}(n)) \in \{(n, \mathcal{D}(n))\}$ do
                for all $l' \in \mathrm{Loc} : (l, l') \in \mathrm{Con}$ do
                    for all $a \in A : \boxed{\langle \mathcal{S}, n, \mathcal{D}(n)\rangle \rightsquigarrow \langle l, l', \mathrm{a}\rangle}$ do
                        $s' = \mathrm{simulateAction}(l, n, \mathcal{D}(n), a, l', s)$
                        $states = states \cup s'$
                    end for
                end for
                if $\mathrm{ELoc}(n) \neq$ nil then
                    for all $l' \in \mathrm{Loc} : (\mathrm{ELoc}(n), l') \in \mathrm{Con}$ do
                        for all $a \in A : \boxed{\langle \mathcal{S}, n, \mathcal{D}(n)\rangle \rightsquigarrow \langle \mathrm{ELoc}(n), l', \mathrm{a}\rangle}$ do
                            $s' = \mathrm{simulateAction}(l, n, \mathcal{D}(n), a, l', s)$
                            $states = states \cup s'$
                        end for
                    end for
                end if
            end for
        end for
    until $states \neq \emptyset$
end function
```

3.5.1 Example

The execution of the algorithm is demonstrated with a small example. The model for this example is depicted in Figure 3.4. The system consist of three locations, an office with a desk and a repository where files and keys are stored. This first part of the algorithm will create the initial state from which the system will evolve to a fixed point. The entire execution of the algorithm is shown in Table 3.1 below.

From the initial state (state 0) the actor can move to the repository which will create a new state. All other actions from initial state will result in transitions that makes a self loop. At the repository the actor can basically perform two action that would change the state. Either he can go back to his office (does not create a new state) or he can input the key. Remark that when he obtains

Algorithm 9 simulateAction(l, n, $\mathcal{D}$(n), a, l′, s)

```
function SIMULATEACTION($l, n, \mathcal{D}(n), a, l', s$)
    if $a \in \{e, elog\}$ then
        if ELoc($n$) = nil then
            ELoc($n$) = $l$
        end if
        $\mathcal{D}'(n) = \mathcal{D}(n) \cup$ decryptKeys($n, \mathcal{D}(n), l'$)
        $s' = s$.remove($n$, $l$).add($n$, $\mathcal{D}'(n)$, $l'$)
        return addState($s, s', n, a, l'$)
    end if
    if $a \in \{m, mlog\}$ then
        if ELoc($n$) $\neq$ nil $\wedge$ Dom(ELoc($n$)) = Dom($l'$) then
            ELoc($n$) = nil
        end if
        $\mathcal{D}'(n) = \mathcal{D}(n) \cup$ decryptKeys($n, \mathcal{D}(n), l'$)
        $s' = s$.remove($n$, $l$).add($n$, $\mathcal{D}'(n)$, $l'$)
        return addState($s, s', n, a, l'$)
    end if
    if $a \in \{o, olog\}$ then
        if $\mathcal{D}(n) \not\subseteq \mathcal{D}(l')$ then
            $s' = s$.add($\mathcal{D}(n) \cup \mathcal{D}(l')$, $l'$)
            return addState($s, s', n, a, l'$)
        else
            $T = T \cup (s \times n\_a\_l' \times s)$                    ▷ selfloop
        end if
    end if
    if $a \in \{i, ilog\}$ then
        if $\mathcal{D}(l') \not\subseteq \mathcal{D}(n)$ then
            $\mathcal{D}'(n) = \mathcal{D}(n) \cup \mathcal{D}(l') \cup$ decryptKeys($n, \mathcal{D}(n) \cup \mathcal{D}(l'), l'$)
            $s' = s$.add($n$, $\mathcal{D}'(n)$, $l$)
            return addState($s, s', n, a, l'$)
        end if
    end if
    return nil
end function
```

Algorithm 10 addstate(s, s′, n, a, l)

```
function ADDSTATE($s, s', n, a, l$)
    $T = T \cup (s \times n\_a\_l \times s')$
    if $s' \notin S$ then
        $S = S \cup s'$
        return $s'$
    else
        return nil
    end if
end function
```

Algorithm 11 decryptKeys(n, κ, l)

```
function DECRYPTKEYS($n, \kappa, l$)
    $k1 = \{\}$                                    ▷ decrypted keys
    $k2 = \{\}$                                    ▷ encrypted keys
    for all $k^\rho \in \kappa$ do
        if $\rho_k \neq \emptyset$ then
            $d'$ = null
            for all $a \in \{d, dlog, i, ilog, o, olog\}$ do
                $d'$ = decrypt($n, l, \kappa, a, k^\rho$)
                if $d' \neq$ null $\wedge\ \rho_{d'} = \emptyset$ then
                    $k1 = k1 \cup d'$
                    break
                end if
            end for
            if $d'$ = null then
                $k2 = k2 \cup d'$
            end if
        end if
    end for
    if $k1 \neq \emptyset$ then
        return $k1 \cup$ decruptKeys($n, k1 \cup k2, l$)
    end if
    return $k1$
end function
```

the key there is no transitions back from state 2 to any previous states. With the key, the actor can now walk into the office. Even that he walk into the same room as before (in state 0), this transition will go into a new state (state 3) as he now have the key which he didn't had before. The key can be placed on the desk which creates state 4. The last state is obtained when the actor moves to the repository. From this state the system can not reach any new states and

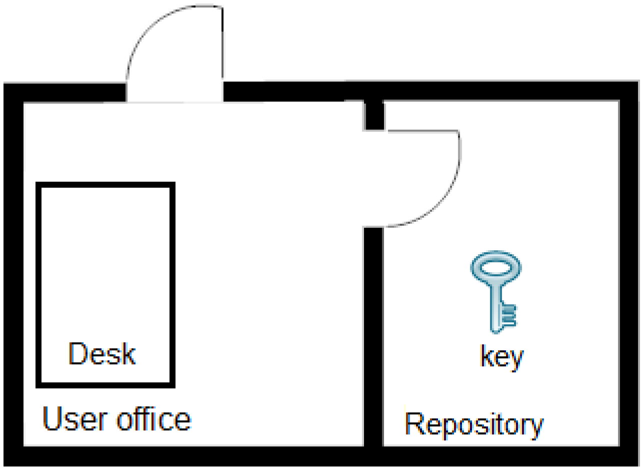


Figure 3.4: Example model for generating LTS

the algorithm stops.

The resulting transition system have 6 states and 24 transitions. Its graphical structure is shown in Figure 3.5

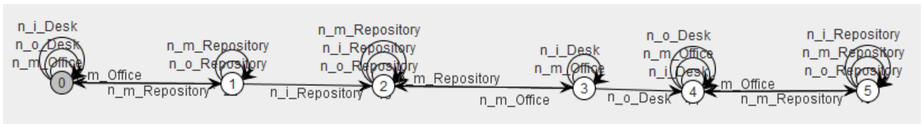


Figure 3.5: The final LTS for the example model in Figure 3.4

3.6 Adding Probabilities

Since the ExASyM model does not have any probabilistic information for an actor performing an action, there is no way to obtain this information directly from the model. So we had to approach this problem differently. The way we have approached this problem was to define 5 interval-classes low, low-medium, medium, medium-high and high with ranges between 1 and 100 in which some of the them overlap. The intervals we have defined are listed below:

- low: [1, 20]
- low-medium: [15, 45]

Iteration	Action	Transitions	State
0	-	-	0 - [{}_desk_{}, {(n_{})}_office_{}, {}_repository_{key}]
1	n_i_desk	0 –> 0	-
1	n_m_office	0 –> 0	-
1	n_o_desk	0 –> 0	-
1	n_m_repository	0 –> 1	1 - [{}_desk_{}, {}_office_{}, {(n_{})}_repository_{key}]
2	n_m_office	1 –> 0	-
2	n_m_repository	1 –> 1	-
2	n_o_repository	1 –> 1	-
2	n_i_repository	1 –> 2	2 - [{}_desk_{}, {}_office_{}, {(n_{key})}_repository_{key}]
3	n_m_repository	2 –> 2	-
3	n_i_repository	2 –> 2	-
3	n_o_repository	2 –> 2	-
3	n_m_office	2 –> 3	3 - [{}_desk_{}, {(n_{key})}_office_{}, {}_repository_{key}]
4	n_m_repository	3 –> 2	-
4	n_i_desk	3 –> 3	-
4	n_m_office	3 –> 3	-
4	n_o_desk	3 –> 4	4 - [{}_desk_{key}, {(n_{key})}_office_{}, {}_repository_{key}]
5	n_o_desk	4 –> 4	-
5	n_i_desk	4 –> 4	-
5	n_i_office	4 –> 4	-
5	n_m_Repository	4 –> 5	5 - [{}_desk_{key}, {}_office_{}, {(n_{key})}_repository_{key}]
6	n_m_office	5 –> 4	-
6	n_i_repository	5 –> 5	-
6	n_m_repository	5 –> 5	-
6	n_o_repository	5 –> 5	-

Table 3.1: Execution sequence of the algorithm for the model in Figure 3.4

- medium: [40, 65]
- medium-high: [50, 80]
- high : [75, 100]

Depending on the action the actor wants to perform, each transition will get a random value from one of these intervals (or an intersection of two intervals). For instance if a transition makes a self loop it will get a value from the low interval since making a self loop does not change the state of the actor and the fact that an actor would probably not perform something repeatable that doesn't change anything. On the other hand, if there is an action that would give the actor new capabilities e.g. input a file that he doesn't have, then this transition would probability have a medium-high to high probability. In this case the interval would be [50-100] and a random score from this interval would be given.

Distributing probability scores in this ways makes it easy to assign probabilities to transitions. Assigning the exact probability would be desired, but obtaining such information would be quite difficult, if not impossible even if one asked the actor himself about what the probability of making a certain action would be.

We have in this project tried to assign probabilities based on the most relevant parameters we could derive from the LTS and the ExASyM model. The parameters that we are currently using are the actions the actor wants to perform and the states from the generated LTS. Among these parameters we could also base the probabilities on actors, data items and locations as these information is available at each state and transition, but these has been left deliberately. More on this will be discussed in section 6.2.
Currently the probability scores for each transition are given based on the six conditions as follows:

- All actions that makes a self loop on the initial state (s0 –> s0) : medium probability score.
- All actions that goes to a new state from initial state (s0 –> A) : low-medium probability score.
- Input or output actions that makes an self loop (A –> A) : low probability score.
- Move or eval actions that makes a self loop (A –> A) : low-medium - medium probability score.
- Move or eval actions that leads to a new state (A –> B) : medium-high probability score.
- Input or output actions that leads to a new state (A –> B) : high probability score.

The function below maps each transition in T to an probability score from an interval above depending on the condition it fulfils:

$$\forall(\text{action}, s, t) \in T : Pr(\text{action}, s, t) \longrightarrow Random(Interval)$$

When each transition has been mapped to a random value from the interval they will be adjusted such that the sum of all outgoing transition from a state sums up to 1. The percentage weight of each transition is found. The algorithm is listed below.

Algorithm 12 adjustProbability(T)

```
1: function adjustProbability($T$)
2:     $sum = 0$
3:     for all $(\text{action}, s, t) \in T$ do
4:         $sum = sum + Pr(\text{action}, s, t)$
5:     end for
6:     for all $(\text{action}, s, t) \in T$ do
7:         $Pr(\text{action}, s, t) = Pr(\text{action}, s, t)/sum$
8:     end for
9: end function
```

3.7 State-Space Analysis

One of the main problems of the LTS that is being generated in this project, is that the state space can potentially grow too large. Currently the state space is: $2^{actors^{data}} \times 2^{locations} \times 2^{data}$.

In order to have every details of the system this large state-space couldn't be avoided. Otherwise some information loss would happen which would result in a faulty LTS. As mentioned in section 3.1.1 there exists many techniques for reducing the state space. However, we have not implemented such a technique in this project.

Chapter 4

The Software Tool

This chapter focuses on the details of the design and implementation of the tool that has been developed for transforming the ExASyM model to a LTS. The tool has been implemented in the Java programming languages using Eclipse on Windows operating system. However the program should be able to run in all platforms that has the Java Virtual Machine (JVM) installed without recompiling the source code.

4.1 User Interface

The user interface is primarily console based on which the most important information is shown. The tool we have developed also consists of a GUI panel that shows the graphical representation of the generated LTS. Finally we make use of the software tool called PRISM that performs the model checking and computes the probability of a given property.

4.1.1 GUI Panel

The GUI panel of the program consist of two parts. The upper part is a graphical visualization of the generated LTS and the lower part is a status panel. The GUI does not have any functionalities as such. It is purely used for visualization purposes. The GUI has been implement such that it will only be shown if there are less than 50 states, otherwise it becomes too large and unmanageable for any good use.

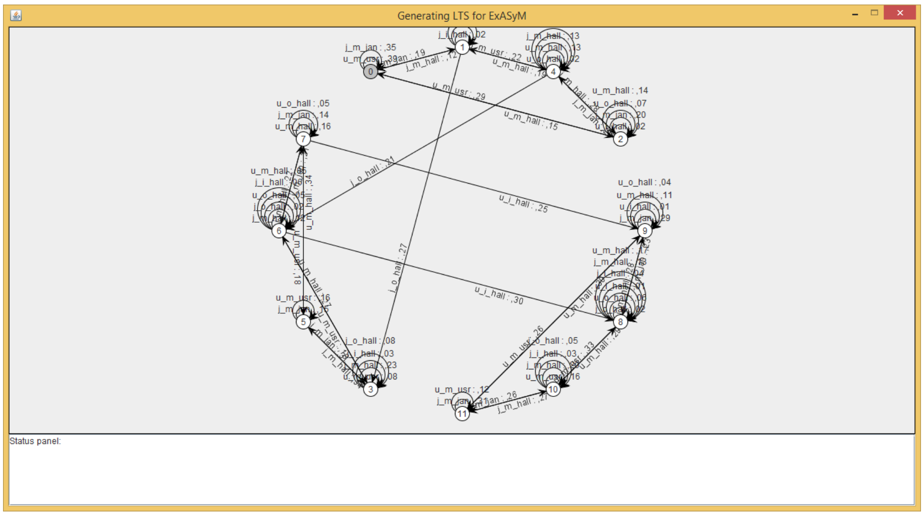


Figure 4.1: The GUI panel of the tool

The GUI provides two modes: translate (pan) mode and picking mode. You can change between translate mode by typing “t” and to picking mode by typing “p”. These two modes activates different functionalities.

The supported functionalities in translate mode are:

1. Left mouse click and mouse drag in the graph window allows you to translate (pan) the graph.
2. Shift, left mouse click and drag in the graph window allows you to rotate the graph.
3. Control, left mouse click and drag in the graph window allows you to shear the graph.
4. Mouse wheel or equivalent allows you to scale (zoom) the graph.

Functionalities in picking mode are:

1. Left mouse click and mouse drag on a node allows you to move the node.

The lower panel is a status panel that prompts detailed information of the state that has been clicked. The status panel has a right click function that provides you to clear the text on the panel.

4.1.2 Console

The console is the essential part of the tool. Basically the program prints 3 sections on the console.

The first section is the textual representation of the ExASyM model in consideration. It shows the details of the system for locations, connections, actors and data. As shown in Figure 4.2 each location consist of 4 parts. The first part is the name of the location, followed by the policies applied to the location, the third part tells the domain the location is belonging to and the last part is a list of data items located at that location. Each data item is shown with its name and the policy restriction applied to it.
A policy has either an actor, a location, a datum, a $\star$ (star) or $\emptyset$ (empty-set) together with a set of allowed actions next to it. This represents which capability an actor must have in order to grant access to the datum or location.

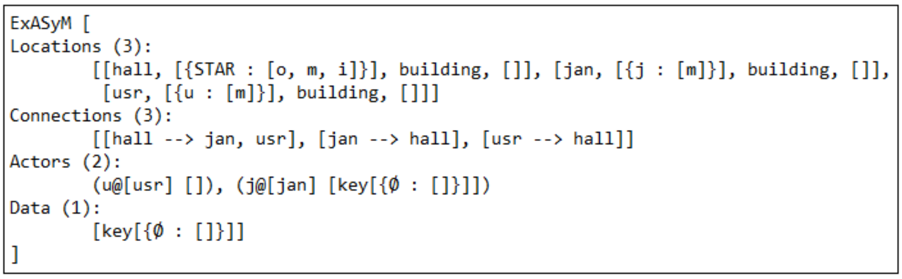
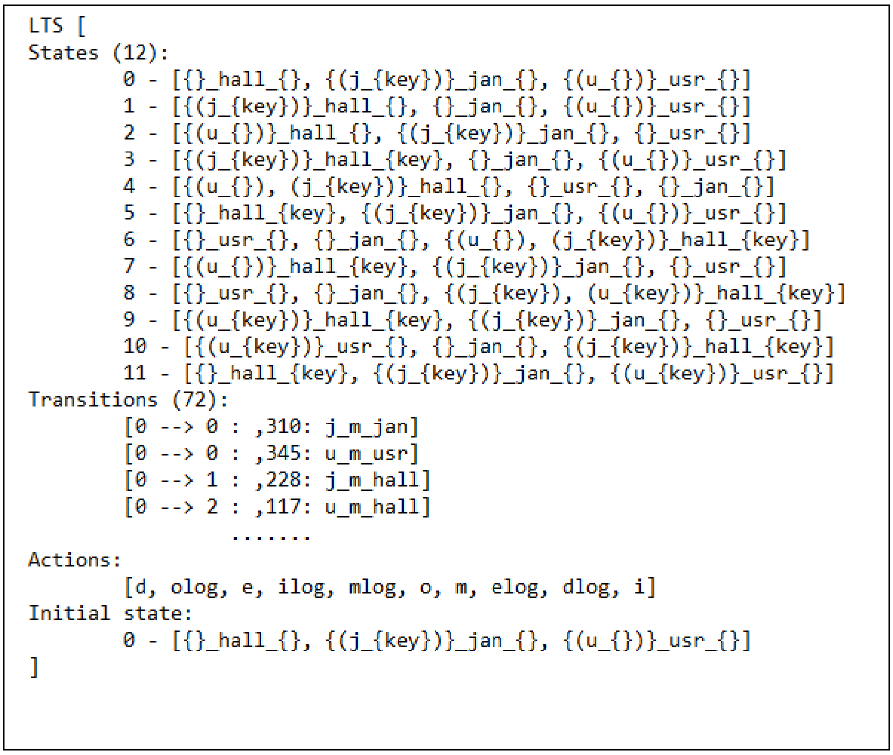

```
ExASyM [
Locations (3):
        [[hall, [{STAR : [o, m, i]}], building, []], [jan, [{j : [m]}], building, []],
         [usr, [{u : [m]}], building, []]]
Connections (3):
        [[hall --> jan, usr], [jan --> hall], [usr --> hall]]
Actors (2):
        (u@[usr] []), (j@[jan] [key[{∅ : []}]])
Data (1):
        [key[{∅ : []}]]
]
```

Figure 4.2: Output of the console section 1

The second part of the console shows the details of the generated LTS. This section consists of four parts.
The first part shows the states that has been generated. Remember that a state consist of a set of valid configurations in the format of $(\{a, \mathcal{D}(a)\}, l, \mathcal{D}(l))$. This represents a location with the data available at that location together with the actors located at l and their data.
The transitions are also shown. Each transition shows the source and target

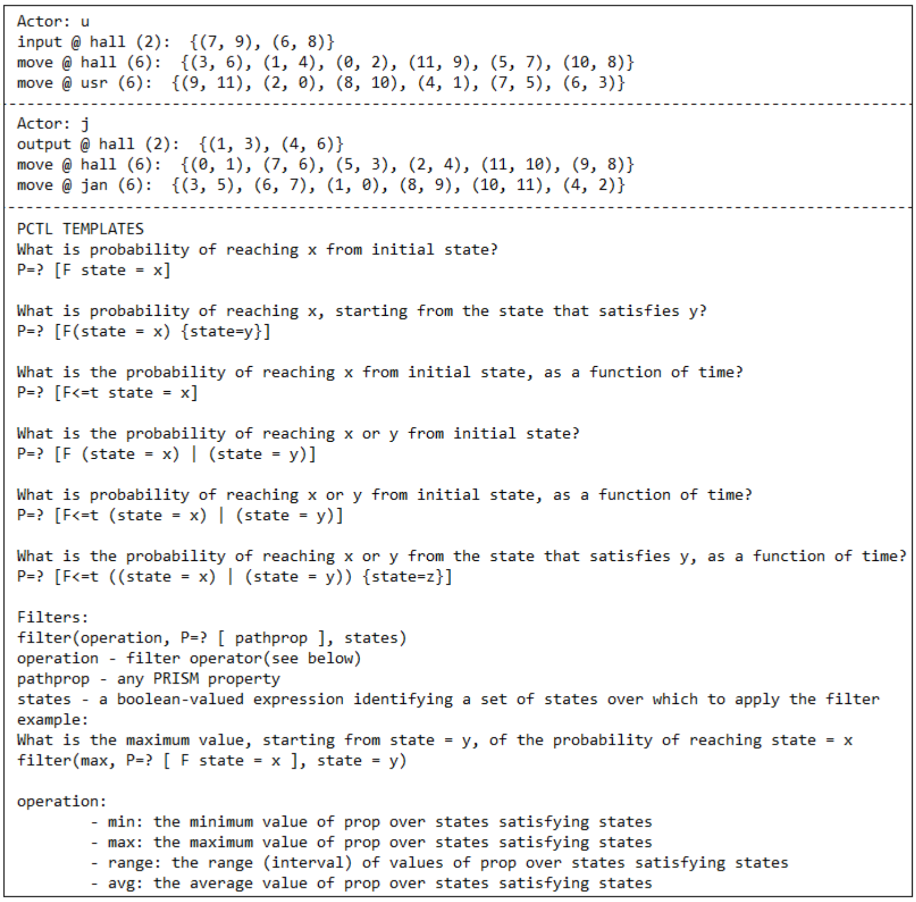
state together with its action and probability. Furthermore the initial state and the set of actions that are supported is shown as the last part of this section. The output of this section is depending on the number of states generated. If more than 50 states are generated the list of transitions will not be shown. However, the transitions can be found in the PRISM file that is generated by the tool. Figure 4.3 shows an output of section 2 on the console.



```
LTS [
States (12):
        0 - [{}_hall_{}, {(j_{key})}_jan_{}, {(u_{})}_usr_{}]
        1 - [{(j_{key})}_hall_{}, {}_jan_{}, {(u_{})}_usr_{}]
        2 - [{(u_{})}_hall_{}, {(j_{key})}_jan_{}, {}_usr_{}]
        3 - [{(j_{key})}_hall_{key}, {}_jan_{}, {(u_{})}_usr_{}]
        4 - [{(u_{}), (j_{key})}_hall_{}, {}_usr_{}, {}_jan_{}]
        5 - [{}_hall_{key}, {(j_{key})}_jan_{}, {(u_{})}_usr_{}]
        6 - [{}_usr_{}, {}_jan_{}, {(u_{}), (j_{key})}_hall_{key}]
        7 - [{(u_{})}_hall_{key}, {(j_{key})}_jan_{}, {}_usr_{}]
        8 - [{}_usr_{}, {}_jan_{}, {(j_{key}), (u_{key})}_hall_{key}]
        9 - [{(u_{key})}_hall_{key}, {(j_{key})}_jan_{}, {}_usr_{}]
        10 - [{(u_{key})}_usr_{}, {}_jan_{}, {(j_{key})}_hall_{key}]
        11 - [{}_hall_{key}, {(j_{key})}_jan_{}, {(u_{key})}_usr_{}]
Transitions (72):
        [0 --> 0 : ,310: j_m_jan]
        [0 --> 0 : ,345: u_m_usr]
        [0 --> 1 : ,228: j_m_hall]
        [0 --> 2 : ,117: u_m_hall]
                 .......
Actions:
        [d, olog, e, ilog, mlog, o, m, elog, dlog, i]
Initial state:
        0 - [{}_hall_{}, {(j_{key})}_jan_{}, {(u_{})}_usr_{}]
]
```

Figure 4.3: Output of the console section 2

The last part of the console shows some interesting states together with a list of PCTL formulae (property specification) that could be interesting for further investigation in the PRISM model checker.
It shows for each actor where and in which state an action was performed. The list gives an quick overview for finding the necessary information needed for specifying a PCTL property that can be carried out by PRISM. The example shown in Figure 4.4 tells us that the actor U is inputting from the hallway in state 9 and 8 that the actor J has outputted in state 3 and 6.



```
Actor: u
input @ hall (2):  {(7, 9), (6, 8)}
move @ hall (6):  {(3, 6), (1, 4), (0, 2), (11, 9), (5, 7), (10, 8)}
move @ usr (6):  {(9, 11), (2, 0), (8, 10), (4, 1), (7, 5), (6, 3)}
-----------------------------------------------------------------------------------------
Actor: j
output @ hall (2):  {(1, 3), (4, 6)}
move @ hall (6):  {(0, 1), (7, 6), (5, 3), (2, 4), (11, 10), (9, 8)}
move @ jan (6):  {(3, 5), (6, 7), (1, 0), (8, 9), (10, 11), (4, 2)}
-----------------------------------------------------------------------------------------
PCTL TEMPLATES
What is probability of reaching x from initial state?
P=? [F state = x]

What is probability of reaching x, starting from the state that satisfies y?
P=? [F(state = x) {state=y}]

What is the probability of reaching x from initial state, as a function of time?
P=? [F<=t state = x]

What is the probability of reaching x or y from initial state?
P=? [F (state = x) | (state = y)]

What is probability of reaching x or y from initial state, as a function of time?
P=? [F<=t (state = x) | (state = y)]

What is the probability of reaching x or y from the state that satisfies y, as a function of time?
P=? [F<=t ((state = x) | (state = y)) {state=z}]

Filters:
filter(operation, P=? [ pathprop ], states)
operation - filter operator(see below)
pathprop - any PRISM property
states - a boolean-valued expression identifying a set of states over which to apply the filter
example:
What is the maximum value, starting from state = y, of the probability of reaching state = x
filter(max, P=? [ F state = x ], state = y)

operation:
        - min: the minimum value of prop over states satisfying states
        - max: the maximum value of prop over states satisfying states
        - range: the range (interval) of values of prop over states satisfying states
        - avg: the average value of prop over states satisfying states
```

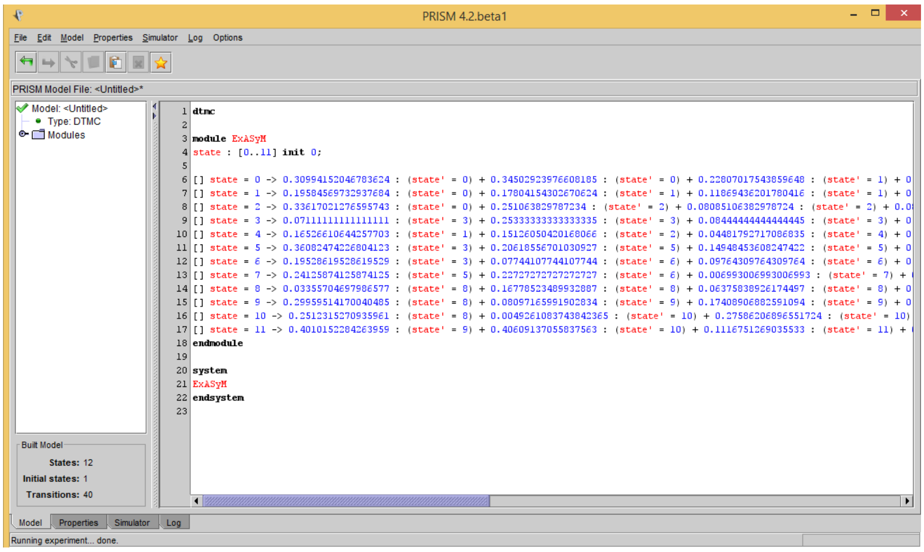
Figure 4.4: Output of the console section 3

4.1.3 Output Files

The software tool that we have developed generates two files that both can be used with PRISM. One of them is a discrete time transition system (without probabilities). It contains the states and transitions, and is suitable for regular model checking. The file has the extension .nm.
The other file is a discrete time Markov chain that contains probabilities attached to the transitions and can be used for probabilistic model checking. Both files can be directly loaded in PRISM and the model checking can be performed.

4.1.4 PRISM

For this project we are using a software tool called PRISM. It is used for performing the model checking on the generated LTS. PRISM is a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour. Models are described using the PRISM language, a simple, state-based language. PRISM provides support for automated analysis of a wide range of quantitative properties of these models. Furthermore it also supports PCTL property specification language. The software is open source and can be downloaded from their homepage.


PRISM 4.2.beta1
File Edit Model Properties Simulator Log Options
PRISM Model File: <Untitled>*
Model: <Untitled>
Type: DTMC
Modules
Model Properties Simulator Log

```
dtmc

module ExASyM
state : [0..11] init 0;

[] state = 0 -> 0.30994152046783624 : (state' = 0) + 0.34502923976608185 : (state' = 0) + 0.22807017543859648 : (state' = 1) + 0
[] state = 1 -> 0.19584569732937684 : (state' = 0) + 0.17804154302670624 : (state' = 1) + 0.11869436201780416 : (state' = 1) + 0
[] state = 2 -> 0.33617021276595743 : (state' = 0) + 0.251063829787234 : (state' = 2) + 0.08085106382978724 : (state' = 2) + 0.0
[] state = 3 -> 0.07111111111111111 : (state' = 3) + 0.25333333333333335 : (state' = 3) + 0.08444444444444445 : (state' = 3) + 0
[] state = 4 -> 0.16526610644257703 : (state' = 1) + 0.15126050420168066 : (state' = 2) + 0.04481792717086835 : (state' = 4) + 0
[] state = 5 -> 0.36082474226804123 : (state' = 3) + 0.20618556701030927 : (state' = 5) + 0.14948453608247422 : (state' = 5) + 0
[] state = 6 -> 0.19528619528619529 : (state' = 3) + 0.07744107744107744 : (state' = 6) + 0.09764309764309764 : (state' = 6) + 0
[] state = 7 -> 0.24125874125874125 : (state' = 5) + 0.22727272727272727 : (state' = 6) + 0.006993006993006993 : (state' = 7) +
[] state = 8 -> 0.03355704697986577 : (state' = 8) + 0.16778523489932887 : (state' = 8) + 0.06375838926174497 : (state' = 8) + 0
[] state = 9 -> 0.29959514170040485 : (state' = 8) + 0.08097165991902834 : (state' = 9) + 0.17408906882591094 : (state' = 9) + 0
[] state = 10 -> 0.251231527093596l : (state' = 8) + 0.0049261083743842365 : (state' = 10) + 0.27586206896551724 : (state' = 10)
[] state = 11 -> 0.4010152284263959 : (state' = 9) + 0.40609137055837563 : (state' = 10) + 0.1116751269035533 : (state' = 11) +
endmodule

system
ExASyM
endsystem
```

Built Model
States: 12
Initial states: 1
Transitions: 40

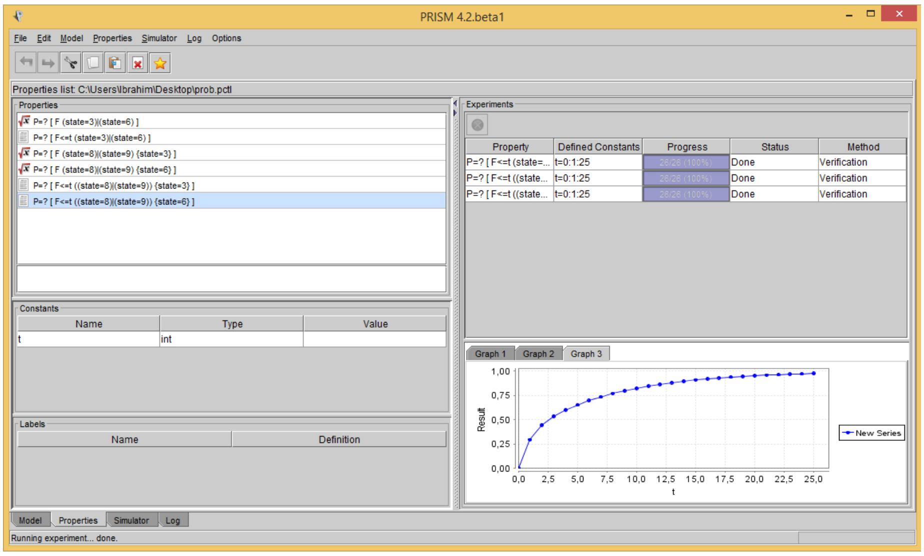
Running experiment... done.

Figure 4.5: The PRISM interface: the Model tab

The tool we have developed in this project provides the model that can be loaded into the model tab so the end-user does not have too much to do there. The interesting part is in the properties tab. This tab allows to write properties that we would like to verify on the model, and invoke the model checker (Figure 4.6). PRISM can also calculate transient state and steady-state distribution.

4.1.5 Loading the Program

The first thing the user of the tool must do is to load the program with a system specification. The system specification has the same syntax as the language presented in Section 2.7. The tool will parse the system specification and respond



Properties list: C:\Users\Ibrahim\Desktop\prob.pctl

```
P=? [ F (state=3)|(state=6) ]
P=? [ F<=t (state=3)|(state=6) ]
P=? [ F (state=8)|(state=9) {state=3} ]
P=? [ F (state=8)|(state=9) {state=6} ]
P=? [ F<=t ((state=8)|(state=9)) {state=3} ]
P=? [ F<=t ((state=8)|(state=9)) {state=6} ]
```

Name	Type	Value
t	int	

Name	Definition

Property	Defined Constants	Progress	Status	Method
P=? [F<=t (state=...	t=0:1:25	26/26 (100%)	Done	Verification
P=? [F<=t ((state...	t=0:1:25	26/26 (100%)	Done	Verification
P=? [F<=t ((state...	t=0:1:25	26/26 (100%)	Done	Verification

Running experiment... done.

Figure 4.6: The PRISM interface: the Properties tab

with an error message if the specification can not be parsed. If the specification can be parsed with no problems, its textual representation will be printed on the screen. The program will thereafter transform the ExASyM model into a LTS. Depending on the size of the model, this can take some time. If there are less than 50 states in the generated LTS a GUI will be shown similar to the one in Figure 4.1.

4.2 Program Source

The program we have developed is written in Java using Eclipse. Java is a widespread object-oriented programming language that has been used since 1995 when it was released by Sun Microsystems. One of the main advantages of Java is its ability to move Java written programs from one computer to another easily.
The reason for choosing Java as our target language is that I have good experience with the language and the project had to be compatible with other program fragments that is developed by my supervisor. But the ideas presented in this report can also be implemented in other language as well.

4.2.1 Program Structure

The source code has been developed as a Java project in the Eclipse IDE. Eclipse provides a lots of features that helps the developer while coding and the ability of customizing the environment with extensible plug-ins is great.

Figure 4.7 shows a package diagram for the source code. The ExASyM package contains classes that implements most of the theory presented in chapter 2. Classes in this package represents the ExASyM model in which actors, data, locations, policies etc... are found. A class diagram for the classes in this package is found in appendix B.1.

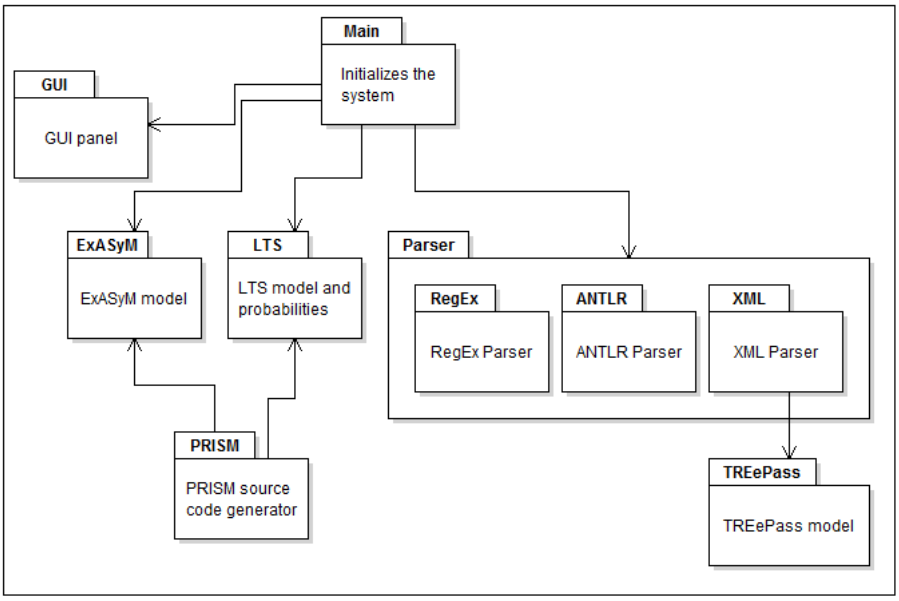


Figure 4.7: Package diagram of the source code

The GUI package contains two classes. One of them implements the status panel and the other one implements the LTS panel. The status panel is a simple `JTextArea` object that is used to print details when a state on the LTS is clicked. The LTS panel is based on the JUNG framework.

The LTS package contains source code that implements the algorithm used for generating the LTS based on the ExASyM model provided by the user. The theory in chapter 3 is implemented in classes found in this package.

The parser package contains an interface to the three parsers that is provided in the project. Each parser creates `ExASyM` object that is suitable for generating the corresponding LTS.
The source code currently supports three parsers and the reason for this, is that the first parser that was developed was based on regular expressions. Regular expressions made it fast to develop a functional parser that could be used in the project. However, regular expression might have its advantages of being easy to write, but it also have a major disadvantage of being fragile to small mistakes in the syntax which can be hard to find and fix. Therefore a new parser was needed. The second parser has been generated using ANTLR (**An**other **T**ool for **L**anguage **R**ecognition). ANTLR is a tool that can generate lexers and parsers for the Java programming language based on a grammar file. The grammar specifies a language and ANTLR generates as output the source code for a recognizer for that language (in Java). The grammar file for the ExASyM model (presented in section 2.7) is found in the ANTLR package.
The third parser is a XML parser that is provided by my supervisor Christian. The classes in the TREsPASS package is used together with the xml parser for an intermediate representation.

The PRISM package contains classes for transforming the LTS into PRISM code. As mentioned before two output files are generated. A PRISM code for a transition system and code for a Markov chain. The source code in this package will also make a simple analysis of the LTS and prints those states and transtions that could be interesting for detailed investigation.

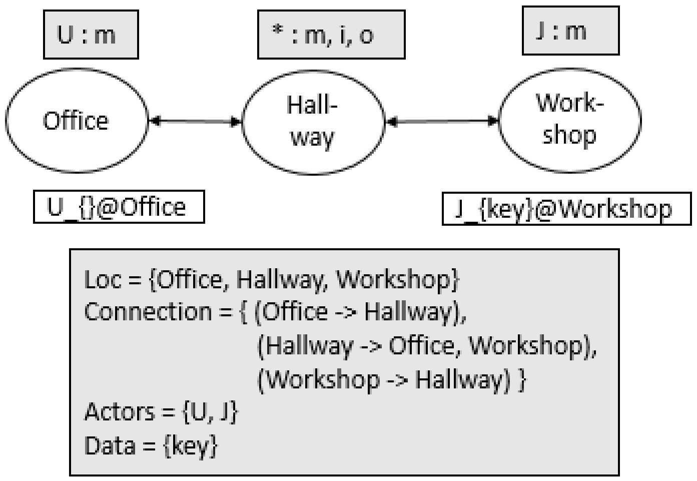
Chapter 5

Analysis

One of the objectives of this project was to develop a tool for performing the insider analysis. A description of the implementation was given in Chapter 4. This chapter will focus on the application on how to use the tool. We will prensent two system specifications for evaluating the tool where each specification will demonstrate important properties. Hopefully this chapter will also demonstrate the correctness of the algorithm, and the implementation of the tool.

5.1 Analysis 1

For the first analysis we are using a simple model in order to show the strengths of transition systems and compute probabilities for some event that could be interesting. The first model that we take under consideration has three rooms, a user's office, a workshop and a hallway interconnecting the rooms together. The system has two actors, a user who is initially located at his office and a janitor who located at his workshop. The user can not enter the workshop and the janitor can not enter the office which is also indicated on the policy restrictions in Figure 5.1. However they can both meet at the hallway in which they can input and output whatever they want. Initially there is one data item in the system which is the key, that is given to the janitor.



Loc = {Office, Hallway, Workshop}
Connection = { (Office -> Hallway),
(Hallway -> Office, Workshop),
(Workshop -> Hallway) }
Actors = {U, J}
Data = {key}

Figure 5.1: The graph for the system model used for analysis 1

The generated LTS for the model above is shown in figure 5.2. It has 12 states in which state 0 (the one marked in grey) is the initial state and 72 transition. Each transition has a probability in the interval]0,1].

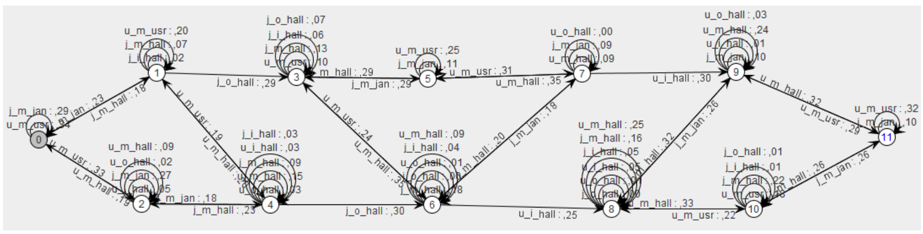


Figure 5.2: The LTS for the system model in analysis 1

The software tool also generates code for a Markov chain that can be used in PRISM. PRISM allows to calculate the transient state distribution in order to see the probabilities of where the process will be after x steps in the future from the initial state. For this example we have set x = 4. In PRISM this can be computed as follows: *model* -> *compute* -> *Transient Probabilities*. Enter 4 in the prompted dialoguebox. Prism outputs following values:

$$\begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \left(.359\right. & .141 & .184 & .083 & .112 & .025 & .069 & .012 & \left..010\right) \end{array}$$

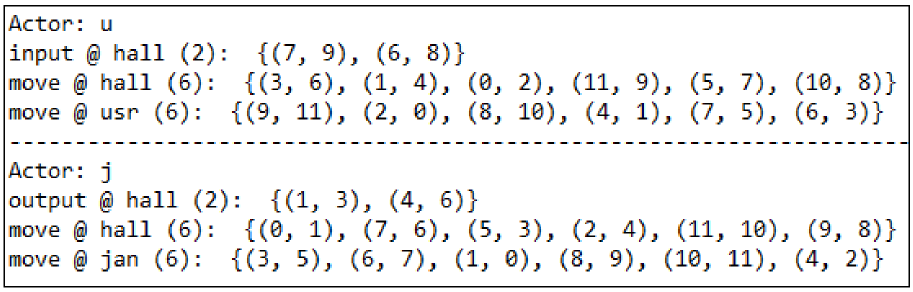
Remark that state 9, 10 and 11 are not reachable from the inital system after 4 steps, thus they are not shown in the vector!

PRISM also has built-in functionaly for computing the steady-state distrubution. This will gives and idea of where the actors will be and what data they will hold in the long run. In PRISM steady-state is calculated as follows: *model* -> *compute* -> *Steady-state probabilities.* Prism outputs following results:

$$\begin{array}{cccc} 8 & 9 & 10 & 11 \\ \left(.316\right. & .245 & .218 & \left..219\right) \end{array}$$

Sometimes PRISM can give an error message while computing the steady-state probabilities. This can be fixed in *Options* -> *Options* -> and select *Gauss-Seidel* in the Linear equations method.
The results we obtain here shows us that the system in the long run will end up somewhere in the last 4 states in the system. The probability of being in state 0-7 is so low (about 0%) that PRISM has omitted these states. The highest probability among the results is being in state 8 with 31.6% probability (`8 - [{}_office_{}, {}_workshop_{}, {(j_{key}), (u_{key})}_hallway_{key}]`). In state 8 we know that the user and the janitor is located at the hallway, both of them have the key and the key is also found at the hallway.

The analysis above gives a holistic view of where the actors will be and how the system will process its execution. However, it does not give any interesting properties of the individual actors. This information can also be obtained. The last part in the console of the tool that we have implemented gives some hint about who has done what and where. The output from tool is shown in Figure 5.3.



```
Actor: u
input @ hall (2):  {(7, 9), (6, 8)}
move @ hall (6):  {(3, 6), (1, 4), (0, 2), (11, 9), (5, 7), (10, 8)}
move @ usr (6):  {(9, 11), (2, 0), (8, 10), (4, 1), (7, 5), (6, 3)}
----------------------------------------------------------------------
Actor: j
output @ hall (2):  {(1, 3), (4, 6)}
move @ hall (6):  {(0, 1), (7, 6), (5, 3), (2, 4), (11, 10), (9, 8)}
move @ jan (6):  {(3, 5), (6, 7), (1, 0), (8, 9), (10, 11), (4, 2)}
```

Figure 5.3: Output from the tool

From these information we can see that the janitor has output something in state 3 and 6 at the hallway and that the user has picked it up at state 8 and 9. What could be interesting here is to find the probability of dropping the key at the hallway by the janitor (as a function of time), and find the probability of picking the key up by the user after it has dropped (as a function of time). For answering these questions we can use some of the PCTL templates that is printed on the console. The PCTL formulae we are interested in are listed in Table 5.1:

PCTL Formulae	Probability
filter(avg, P=? [F (state = 3) | (state = 6)], state = 0)	1
filter(avg, P=? [F (state = 8) | (state = 9)], state = 3 | state = 6)	1
filter(range, P=? [F<=t (state = 3) | (state = 6)], state = 0)	Fig 5.4
filter(avg, P=? [F<=t (state = 3) | (state = 6)], state = 0)	Fig 5.4
filter(range, P=? [F<=t (state = 8) | (state = 9)], state = 3 | state = 6)	Fig 5.5
filter(avg, P=? [F<=t (state = 8) | (state = 9)], state = 3 | state = 6)	Fig 5.5

Table 5.1: PCTL formulae for analysis 1

The probabilities for the first two properties is 1. This was expected as the system tends to go the to last four states in the system in the long run, meaning that these actions are most likely happen.
The graphs that has been generated shows the stepwise progression of the probabilities growing very rapidly. Only after 10 time-steps the janitor has output the key, the probability that the user will pick it up is about 75% and goes to 100% in the long run.
From the obtained results and analysis we can conclude that it can be expected that in this system model the janitor will output the key and that the user will pick it up.

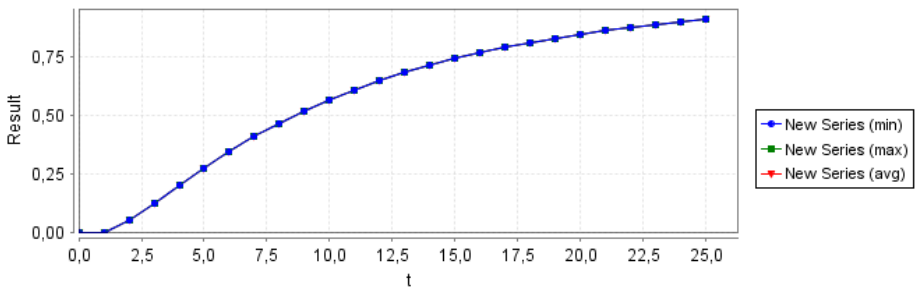


Figure 5.4: The probability as function of time the janitor will output the key at hallway

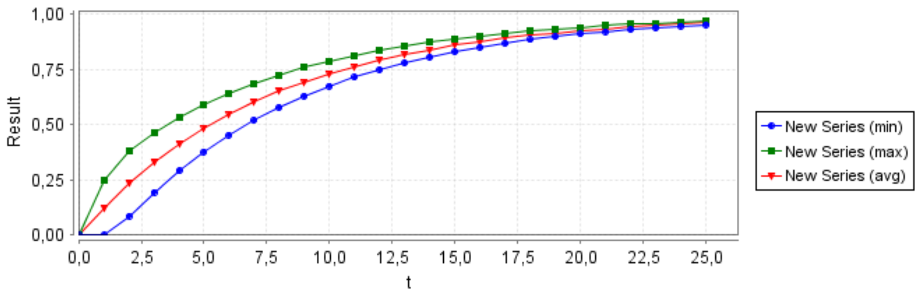


Figure 5.5: The probability as function of time the user will pick up the key after the janitor dropped the key

5.1.1 Use of CEGAR

In section 3.1.1 we discussed CEGAR and how it can be used for reducing the number of states in a transition system. In this section we will show how to apply the technique on a concrete example, on the LTS that was generated by the algorithm in section 3.5.

Notice that from state 3 and 6 (states where the janitor has output the key at hallway) there are no transitions back to the previous states and also there are no transitions from state 7 and 9 back to the lower numbered states. This gives two cuts that could be made such that the states in each section could be clustered into set of states. The resulting transition system have 3 (instead of 12) states and much lower number of transitions. Figure 5.6 shows the cuts and the clustered states. The idea behind this is that we have abstracted the exact location of where the actor might be and focused on where the key is. In state number 1 we know that the actors are at there initial location or at the locations that they can reach and only the janitor knows the key. In state 2, the janitor has output the key at the hallway, but the user has not picked it up yet. He might be at the hallway since everyone has the move access to the location, but we know for sure that he has not picked it up. In state 3 we obtain the information that the user has picked the key up as there are no transitions back to the previous state and that the actors in system are located somewhere in a location they can reach.

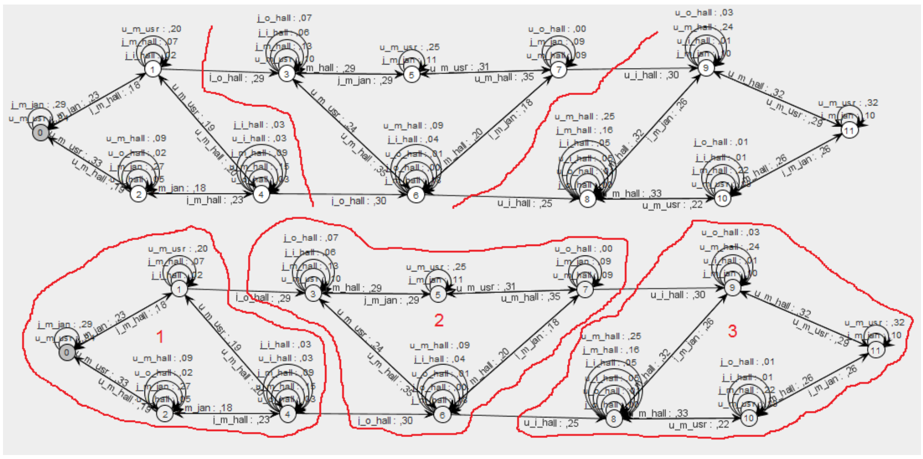


Figure 5.6: Cuts in the graph and abstraction refinement of states

5.2 Analysis 2

For this analysis, we will investigate a simplified version of the running example that was presented in section 2.3. We had to simplify the model as the original have more than 27000 states and 185000 transitions. My computer with Intel Core i7-3517U@1.90GHz processor and 8GB RAM couldn't finish the transformation after 10 minutes of runtime.
In the simplified version of the model, we have removed the receptionist and the basement, but the waste bin and the vault has been moved upstairs to the hallway. The locks on the doors are also removed, although access policies on the locations should restrict the movement of the actors such that no one else than the user can enter his office and the no one else than the janitor can enter his workshop. The vault has one key (restriction) instead of two and the user is the only actor who has the key (CL_v) to enter the vault initially. The refined model shown in Figure 5.7, it is larger then the model presented in the previous section, thus it will have many more states and transitions than before. How-

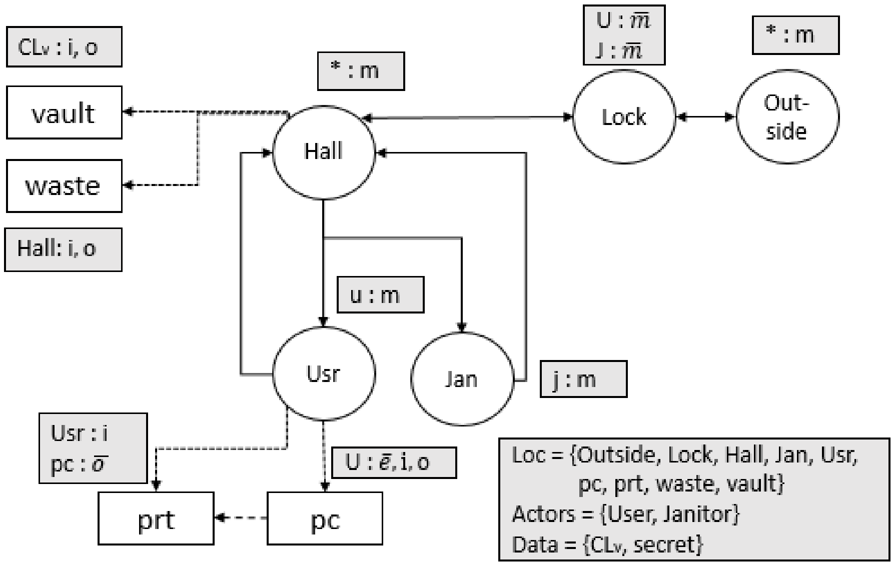


Figure 5.7: The graph for the system model used for analysis 2

ever, with the use of the tool and PRISM we will still be able to derive any probabilistic information we want from the model.

The generated LTS has 1488 and 11692 transitions. This huge state-space is of course a problem, but it should not be a limitation for doing any model checking. The first property we would like to investigate is the progression of the system and how the system will behave in the long run. This can be computed by calculating the steady-state probability for the Markov chain. The vector below shows the top five steady-states probabilities.

$$\begin{array}{ccccc} 1193 & 1368 & 1367 & 191 & 343 \\ \end{array}$$
$$\begin{pmatrix} .184 & .126 & .120 & .108 & .094 \end{pmatrix}$$

```
1193 - [{(j_{secret, clv})}_hall_{}, {}_prt_{secret, clv}, {}_outside_{},
{}_lock_{}, {(u_{secret, clv})}_pc_{secret, clv}, {}_waste_{secret,
clv}, {}_usr_{}, {}_vault_{secret, clv}, {}_jan_{}]

1368 - [{}_hall_{}, {}_prt_{secret, clv}, {}_outside_{}, {}_lock_{},
{(u_{secret, clv})}_pc_{secret, clv}, _waste_{secret, clv}, {}_usr_{},
{}_vault_{secret, clv}, {(j_{secret, clv})}_jan_{}]
```

```
1367 - [{}_hall_{}, {}_prt_{secret, clv}, {}_outside_{}, {(j_{secret,
clv})}_lock_{}, {(u_{secret, clv})}_pc_{secret, clv}, {}_waste_{secret,
clv}, {}_usr_{}, {}_vault_{secret, clv}, {}_jan_{}]

191 - [{}_hall_{}, {}_prt_{clv}, {}_outside_{}, {(j_{})}_lock_{}, {(u_
{clv})}_pc_{clv}, {}_waste_{}, {}_usr_{}, {}_vault_{secret}, {}_jan_{}]

343 - [{}_hall_{}, {}_prt_{clv}, {(j_{})}_outside_{}, {}_lock_{}, {(u_
{clv})}_pc_{clv}, {}_waste_{}, {}_usr_{}, {}_vault_{secret}, {}_jan_{}]
```

With 18.4% probability the system will go into a state where the janitor is located at the hallway and the user is working at his office on the pc. The user has uploaded the key and the secret to the pc and printed those on the printer. The vault has been accesses, as the secret is found everywhere possible, even the janitor got a copy of the key and the secret. Furthermore we can see that the janitor is more active than the user, as he can be found either at the hallway, at the entrance to the office (lock), outside or at his workshop, while the user is most likely working on his pc. Notice that state 191 and 343 are the only two states where the secret has not been taken from the vault and the key has not been thrown out to the waste bin. Due to this behaviour the janitor has not been able to get any data. The likelihood for entering these two states in the long run are 10.8% and 9.4% respectively.

The result above may reflect that the policy restrictions in the system are not so good at protecting access to the vault and the data items. There is a possibility that the janitor can access the data items available in the system in two ways. Either the user outputs the key to the vault, such that the janitor can pick it up from the waste bin and access the secret from the vault, or if the user outputs both data items to the waste bin after he has input the secret from the vault. A serious insider threat exists if the janitor somehow takes these two data items and moves outside. Therefore we would like to know the probability for these two events and the likelihood for moving outside once the janitor got these data items. The last property we would like to find is the probability the janitor moves outside with the key and the secret which he has obtained somehow from the initial state. Thus we have 3 paths we would like to find the probabilities for. Figure 5.8 depicts the paths we are interested in.

The PCTL formulae for the properties we would like to investigate are composed with use of the hints given by the tool. Remark that these states just shows what an actor does and where he does it. It does not show any information about the data items they hold, input or output. For instance there are 112 states where the user outputs to the waste. In 16 of these states the key is output and for the remaining (96) states the key and the secret is output. This information is currently obtained manually by looking through all the 112 states

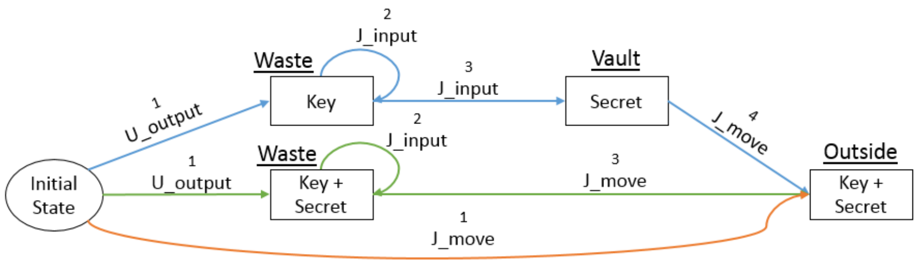


Figure 5.8: Three possibilities (paths) the janitor moves outside with the key and secret

and find those which are relevant to compose the formalae. The PCTL formulae for the analysis in this section is not included in the report, they can be found in the uploaded archive (PRISM/Analysis2/PCTL.pctl). In this section we will just write the properties we are interesting in with words.

The first path consists of four formulae we would like to verify:

(a) What is the probability the user outputs the key in waste bin (from initial state)?

(b) What is the probability the janitor finds the key to the vault after the user has output the key to the waste bin?

(c) What is the probability the janitor inputs from the vault after he finds the key in the waste bin?

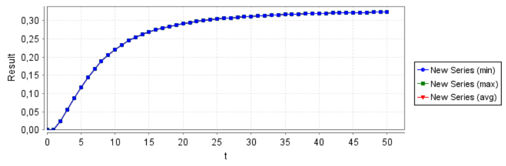
(d) What is the probability the janitor goes outside with the secret and key after he has input from vault?

The answers for these properties are found in Table 5.2

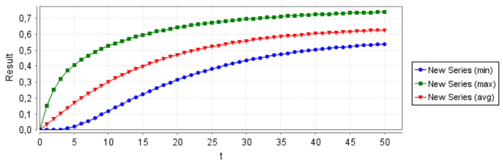
property	min.	avg.	max.	graph
a	32.6%	32.6%	32.6%	Fig 5.9a
b	57.5%	65.1%	75.5%	Fig 5.9b
c	100%	100%	100%	Fig 5.9c
d	100%	100%	100%	Fig 5.9d
total:	18.7%	21.2%	24.6%	

Table 5.2: Results for PCTL formulae for analysis 2, path 1

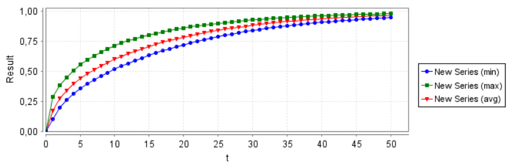
The second path consists of three formulae:



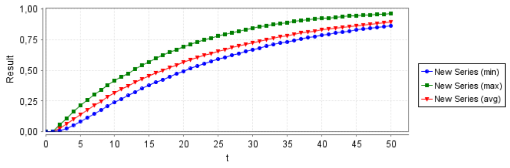
(a)



(b)



(c)



(d)

Figure 5.9: plots of PCTL formulae, path 1

(a) What is the probability the user outputs the key and the secret in waste bin (from initial state)?

(b) What is the probability the janitor inputs the secret and the key after user has output to waste bin?

(c) What is the probability the janitor goes outside with the secret and key after he has input from waste?

The results are showin in Table 5.3

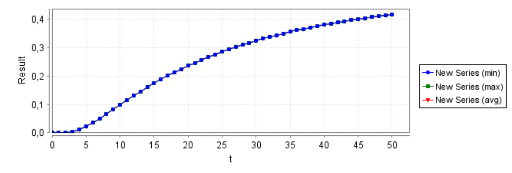
property	min.	avg.	max.	graph
a	46.9%	46.9%	46.9%	Fig 5.10a
b	100%	100%	100%	Fig 5.10b
c	100%	100%	100%	Fig 5.10c
total:	46.9%	46.9%	46.9%	

Table 5.3: Results for PCTL formulae for analysis 2, path 2

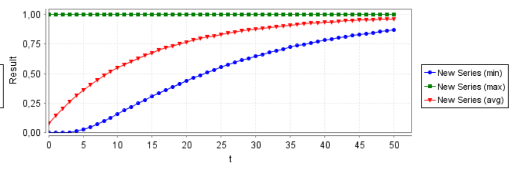
The last path consists of one formulae we would like to answer:

(a) What is the probability the janitor goes outside with the secret and key, as a function of time (from initial state)?

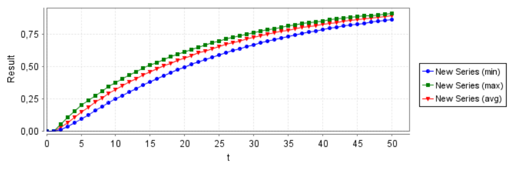
The result for this property is: min: 56.9% avg: 56.9% max: 56.9% Figure 5.11 shows the graph for the probability of the property as a function of time.



(a)



(b)



(c)

Figure 5.10: plots of PCTL formulae, path 2

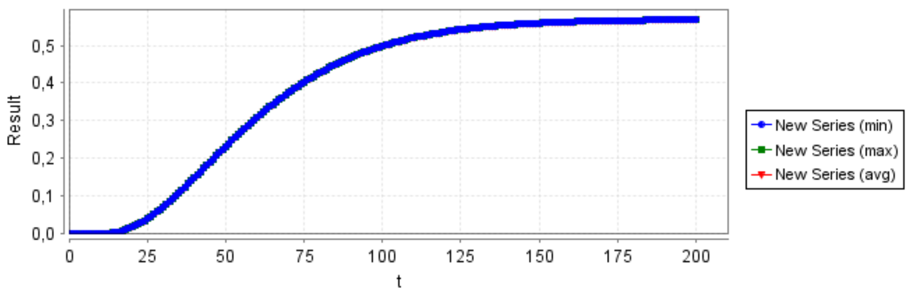


Figure 5.11: plot of PCTL formulae, path 3

The probability that the actor will access both data items and move outside somehow from the initial state of the system is at most 56.9%. In the beginning of the system the likelihood for this event is happening is quite low but after 25 time steps the probability grows up to 55% at time 125, then the probability stabilizes to 56.9%. There are of course many ways that the janitor can obtain the key and the secret and we have investigated three paths that are possible. The results and graphs are shown above. From the results above we can conclude that once the janitor gets to know the data items available in the system, he is most likely to move outside. A way to avoid/reduce the insider threat could be to remove the waste bin to another location where the janitor cannot reach.

Chapter 6

Conclusion

In the previous chapters we have presented the ExASyM framework for doing insider analysis. The main issue of this framework was the lack of measuring probabilities for attack that could strike the system under consideration. We have in this thesis presented an approach that transforms the ExASyM model into a labelled transition system (LTS) that makes it possible to ask this kind of probabilistic questions. Furthermore we have designed and programmed a tool that implements the elements of the framework and the LTS transformation algorithm. We have also evaluated our tool in different use case scenarios in order to show the application of our tool to measure probabilities for attacks that could happen in an organization.

6.1 Achievements

One of the main issues in information security is known as the insider attack. This is an attack type that is performed by users who is already authorized access to a system. Thus it is very hard to detect these kind of attacks as access to data assets are performed permissible.
A number of tools has been presented, ExASyM [PH08], Portunes [TD10], and ANKH [Pie11] that tries to solve these kind of problems. Among these models we have in this thesis focused on the model called ExASyM (An **Ex**tensible

Analysable **Sy**stem **M**odel). The ExASyM model allows the end-user to model an organization's infrastructure and actors, and through a number of analysis finds different kinds of threats that exists in the organization. The details of the model has been covered in chapter 2.
The lack of getting any probabilistic information for these models have not been covered until now. This implies that it was not possible to get any quantitative probabilistic results for measuring how much risk there is for a certain attack will strike the organization and by whom, with the current tools available. In chapter 3 we have introduced the concept of Model Checking and how we could transform the ExASyM into a LTS that is suitable for model checking. With use of model checking we were able to acquire the probabilistic values for attacks which we have demonstrated with examples in chapter 5.

A tool has been developed that transforms the ExASyM model into a LTS. Furthermore the tool helps finding critically vulnerable states for each actor. These states can be used as hints to help the investigator to find probabilities for a certain event to occur. The tool generates two files, that can be used with software tool called PRISM in order to compute probabilities for reaching a certain state. The details of both tools are found in chapter 4.
We believe that we have succeeded in developing an extension to the ExASyM framework for transforming the model into a LTS and hope that concepts presented in this thesis can be used with other frameworks as well, since they are quite similar to each other.

6.2 Discussion and Limitations

Our tool has some limitations. The main limitation to take in consideration is the use of discrete time versus continuous time. A better choice would have been made if we used a transition system where the underlying time domain was continuous (e.g. continuous time Markov chain). Since we didn't had any knowledge about these type of transition system, we have in this thesis used discrete time transition systems and Markov chains.
A second parameter that could have been taking into account in the continuous time transition system is the time it takes to perform an action. This timing information may be different for two actors, even if they performed the same action. A model that incorporates this timing information could be interesting to measure and it would have a great impact on the reality of the framework we have developed.

As discussed in section 3.6 adding the probabilities to the transition are currently based on few input parameters. These could easily be changed such that they become more realistic, if there was a way to relate or differentiate between two locations, data items and actors. E.g. a weight or value could be introduced for locations and data, such that higher value indicates more important location or data item. In this way transitions that goes to a higher valued location or data item would have a higher probability than a lower valued location/data item. Currently each of these elements are equally important and does not impact overall the probability.

The algorithm developed in this thesis is quite pessimistic as it assumes that what can happen will happen. This also leads to the a very large state-space even for small applications. It is pretty unrealistic as it also assumes that every actor will give away all their data, which is not realistic in all cases.

We are quite sure about that these information is critical when modelling IT-systems that simulates real-world application in order to the correctness of the tool and model. Had these information been available the output of this tool and thesis would be much better and realistic.

6.3 Prospective work

The framework can be extended endlessly to make it more realistic and more flexible. In this section we will mention some of the important extension we think that should be worked upon.

The number of states is not a limitation as such, but it is very large due to having complete information about locations, the data at these locations together with the actors and their data. As discuses in the report, a way to reduce the number of states could be achieved with use of CEGAR. We have in this thesis shown an example on how it could be applied in order to reduce the number of states.

The ExASyM model could also be extended with some of the issues mentioned in the discussion section above. The output of the tool would be much more realistic and usable for real-world applications if these limitations could be implemented in the framework.

Appendix A

Calculations

A.1 Transient state

This is the original transition system represented as a matrix and the initial state distribution vector:

$$M = \begin{array}{c|ccccccccc} & Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ \hline Jan & .7 & .3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ Hall & 0 & .1 & .4 & 0 & 0 & .2 & 0 & 0 & .3 \\ Bsmt & 0 & .15 & .15 & .3 & .4 & 0 & 0 & 0 & 0 \\ waste & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ vault & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ FR & 0 & .5 & 0 & 0 & 0 & 0 & .5 & 0 & 0 \\ Rec & 0 & 0 & 0 & 0 & 0 & .6 & .25 & .15 & 0 \\ Out & 0 & 0 & 0 & 0 & 0 & 0 & .65 & .35 & 0 \\ L_{jan} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\iota_{init} = \begin{array}{ccccccccc} Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

We first calculate the M^5 and then multiply the initial distribution vector ι_{init}. $M^5 = M \cdot M \cdot M \cdot M \cdot M$ this can be computed with standard matrix multiplication. We omit the calculations, the result of M^5 is here:

$$M^5 = \begin{array}{c|ccccccccc|} & Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ \hline Jan & .328 & .191 & .185 & .056 & .075 & .057 & .041 & .006 & .057 \\ Hall & .192 & .134 & .355 & .0531 & .070 & .076 & .046 & .013 & .055 \\ Bsmt & .070 & .133 & .288 & .191 & .255 & .017 & .021 & .001 & .019 \\ waste & .042 & .066 & .637 & .071 & .094 & .036 & .007 & .002 & .040 \\ vault & .042 & .066 & .637 & .071 & .094 & .036 & .007 & .002 & .040 \\ FR & .154 & .191 & .118 & .072 & .096 & .108 & .180 & .041 & .035 \\ Rec & .094 & .140 & .171 & .018 & .024 & .216 & .206 & .069 & .058 \\ Out & .058 & .175 & .066 & .023 & .031 & .215 & .301 & .086 & .040 \\ L_{jan} & .375 & .192 & .188 & .034 & .045 & .061 & .031 & .004 & .065 \end{array}$$

We can now multiply the distribution vector ι_{init} with M^5 and we will get the probability distribution of where the janitor might be located after 5 steps. The results are shown below:

$$\mathbf{v} = \begin{array}{ccccccccc} Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ (.328 & .191 & .185 & .056 & .075 & .057 & .041 & .006 & .057) \end{array}$$

A.2 Steady-state

In order to calculate the steady-state probabilities one has to solve the equation:

$$\mathbf{v} \cdot \mathbf{P} = \mathbf{v}$$

It should be remembered that the sum of the vector **v** has to add up to 1.

$$\sum_{i=0}^{n-1} \mathbf{v}(i) = 1$$

The vector **v** is the unknown, and we are solving the equation set with respect to **v**. From this we can line-up the equations to be solved and we obtain the following equation system:

$$\begin{array}{ll} A = 0.7A + I & F = 0.2B + 0.6G \\ B = 0.3A + 0.1B + 0.15C + 0.5F & G = 0.5F + 0.25G + 0.65H \\ C = 0.4B + 0.15C + D + E & H = 0.15G + 0.35H \\ D = 0.3C & I = 0.3B \\ E = 0.4C & A+B+C+D+E+F+G+H+I = 1 \end{array}$$

Using some basic mathematical rules one can isolate B and obtain following results:

$B + B + \frac{8}{3}B + 0.8B + \frac{16}{15}B + 0.4B + \frac{1}{3}B + \frac{1}{39}B + 0.3B = 1 \Leftrightarrow$

$7.6435B = 1 \Leftrightarrow$

$B = \frac{1}{7.6435}$

Now we can obtain the other unknowns of vector $\mathbf{v}$.

$A = \frac{1}{7.6435} \approx 0.1308$

$B = \frac{1}{7.6435} \approx 0.1308$

$C = \frac{8}{3} \cdot \frac{1}{7.6435} \approx 0.3488$

$D = 0.8 \cdot \frac{1}{7.6435} \approx 0.1046$

$E = \frac{16}{15} \cdot \frac{1}{7.6435} \approx 0.1395$

$F = 0.4 \cdot \frac{1}{7.6435} \approx 0.0523$

$G = \frac{1}{3} \cdot \frac{1}{7.6435} \approx 0.0436$

$H = \frac{3}{39} \cdot \frac{1}{7.6435} \approx 0.0100$

$I = 0.3 \cdot \frac{1}{7.6435} \approx 0.0392$

Finally we can collect these values into the resulting steady-state vector. Here is the final results:

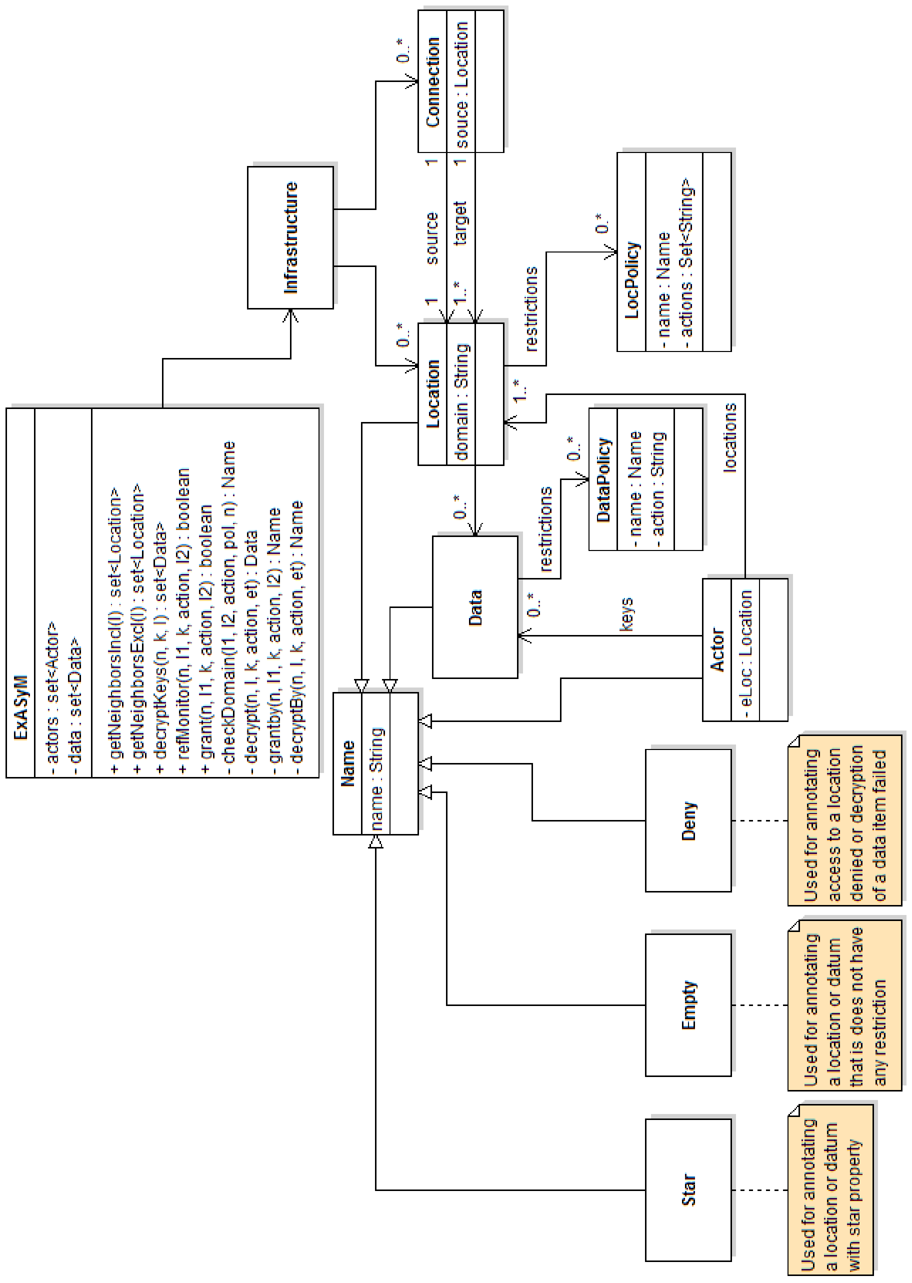
$$\mathbf{v} = \begin{array}{ccccccccc} Jan & Hall & Bsmt & waste & vault & FR & Rec & Out & L_{jan} \\ \big(.130 & .130 & .348 & .104 & .139 & .052 & .043 & .010 & .039 \big) \end{array}$$

Appendix B

Diagrams

B.1 Class diagram

The figure below shows a class digram of the source code in the ExASyM package for the tool that has been developed.


ExASyM
- actors : set<Actor>
- data : set<Data>
+ getNeighborsIncl(l) : set<Location>
+ getNeighborsExcl(l) : set<Location>
+ decryptKeys(n, k, l) : set<Data>
+ refMonitor(n, l1, k, action, l2) : boolean
+ grant(n, l1, k, action, l2) : boolean
- checkDomain(l1, l2, action, pol, n) : Name
- decrypt(n, l, k, action, et) : Data
- grantby(n, l1, k, action, l2) : Name
- decryptBy(n, l, k, action, et) : Name
Infrastructure
Connection
souce : Location
0..*
1
source
1
1
target
1..*
0..*
Location
domain : String
restrictions
0..*
LocPolicy
- name : Name
- actions : Set<String>
1..*
locations
0..*
DataPolicy
- name : Name
- action : String
0..*
restrictions
Data
0..*
keys
Actor
- eLoc : Location
Name
name : String
Deny
Used for annotating access to a location denied or decryption of a data item failed
Empty
Used for annotating a location or datum that is does not have any restriction
Star
Used for annotating a location or datum with star property

Bibliography

[BBN$^+$03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In *GLOBAL COMPUTING: PROGRAMMING ENVIRONMENTS, LANGUAGES, SECURITY AND ANALYSIS OF SYSTEMS, VOLUME 2874 OF LNCS*, pages 88–150. Springer-Verlag, 2003.

[Bis05] M. Bishop. The insider problem revisited. *Proc. of New Security Paradigms Workshop 2005, Lake Arrowhead, CA, USA, Septenber 2005. ACM Press, NewYork*, 2005.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[CWPN06] R. R. Hansen C. W. Probst and F. Nielson. Where can an insider attack? *Workshop on formal aspects in security and trust (FAST 2006)*, 2006.

[GP03] Gorla and Pugliese. Resource access and mobility control with dynamic privileges acquisition. *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[Gun07] Dagur Gunnarson. Static analysis of the insider problem. Master's thesis, Technical University of Denmark, DTU, 2007.

[Her13] Holger Hermanns. Slides from lecuture 9, 02246 model check; discrete time markov chain model checking. Technical University of Denmark, 2013.

[HPN06] René Rydhof Hansen, Christian W. Probst, and Flemming Nielson. *Sandboxing in myKlaim.* IEEE, 2006.

[Iva14] Marieta G. Ivanova. Probabilistic reachability of vulnerabilities in organisations. *Department of Applied Mathematics and Computer Science, Technical University of Denmark - DTU*, 2014.

[PH08] C. W. Probst and R. R. Hansen. An extensible analysable system model. *Elsevier, Information Security Technical Report*, pages 235–246, 2008.

[PH09] C. W. Probst and R. R. Hansen. Analysing access control specifications. *2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 22–33, 2009.

[Pie11] W. Pieters. Representing humans in system security models: An actor-network approach. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, pages 75–92, 2011.

[RA05] R.C Brackney R.H. Anderson. Understanding the insider threat. *RAND Corporation, Santa Monica, CA, U.S.A., March 2005*, 2005.

[RDNP98] G. Ferrari R. D. Nicola and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, pages 315—-330, 1998.

[TD10] P. Hartel T. Dimkov, W. Pieters. Portunes: representing attack scenarios spanning through the physical, digital and social domain. *Proceedings of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'10)*, pages 235–246, 2010.