Vulnerability Analysis of Cyber Security Modelling Language models using Probabilistic Logic

Rick Hindriks
h.n.hindriks@student.utwente.nl

Tuesday 6th December, 2016
MSc Thesis - Final (r3452)

Supervisors

Arend Rensink	University of Twente
Mariëlle Stoelinga	University of Twente
Frank Fransen	Netherlands Organisation for Applied Scientific Research TNO

ABSTRACT

Computer systems are an essential asset of large companies such as banks, financial institutions, utility companies and telecommunication providers. Given their important roles for the functioning of society, these companies are under a constant threat of cyberattacks. Enterprises rely on the availability of these complex ICT systems for their day-to-day operations, and disruptions in the availability of these systems can have disastrous consequences. Given the growing complexity of the attacks and the growing size of network infrastructures, security experts require the use of automated tools to determine the security of their systems. To this end, we propose an automated method for the analysis of vulnerabilities within network architectures, based on the Cyber Security Modelling Language[35] (CySeMoL). We aim to improve the time required for inferring the likelihood of a successful cyberattack in a given network infrastructure, based on the threat model defined by CySeMoL. We define an alternative implementation of the vulnerability analysis using Probabilistic Logic[17] (ProbLog). By using a model-based approach to the analysis of CySeMoL, we provide an extensible method for the development of such an alternative analysis. We have succeeded in achieving this by using intermediate models which capture the threat model of CySeMoL and the definition of concrete network infrastructures. However, our measurements show that the proposed analysis method using ProbLog does not perform better than CySeMoL for larger models.

Keywords

Threat modelling, dynamic risk assessment, SEGRID, CySeMoL, P^{2}AMF, attack trees, vulnerability analysis, probabilistic logic, Prolog, ProbLog, probabilistic inference, weighted model counting, model driven engineering, Eclipse Epsilon, model transformation, model-to-model transformation, model-to-text transformation.

Preface

> *“Then, one day, a student who had been left to sweep up the lab after a particularly unsuccessful party found himself reasoning this way: If, he thought to himself, such a [Infinite Improbability Drive] is a virtual impossibility, then it must logically be a finite improbability. So all I have to do in order to make one is to work out exactly how improbable it is, feed that figure into the finite improbability generator, give it a fresh cup of really hot tea . . . and turn it on! He did this, and was rather startled to discover that he had managed to create the long sought after golden Infinite Improbability generator out of thin air.”*
>
> Discovery of the Infinite Improbability generator — Douglas Adams

This is the thesis for the final project of the *Computer Science* master programme with a specialization in *Methods and Tools for Verification* at the University of Twente. After a ‘research topics’ phase of 10 ECTS, which resulted in a research proposal, the thesis was written during a time period of 30 ECTS at TNO (Netherlands Organisation for Applied Scientific Research).

The goals of this research were inspired by the requirements of the SEGRID (Security for Smart electricity GRIDs) collaboration project. The SEGRID FP7 project was established in 2014 with the goal of enhancing the protection of smart grids against cyberattacks[89]. The broader scope where the research presented in this thesis belongs to, concerns the automated assessment and prediction of security in computer networks.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Research Goals . . . 2
1.4 Approach . . . 3
1.5 Validation . . . 4
1.6 Structure . . . 5
1.7 List of abbreviations . . . 6

2 Background 7
2.1 Introduction . . . 7
2.2 Threat Modelling . . . 8
2.3 The Cyber Security Modelling Language . . . 10
2.4 Model Driven Engineering . . . 20
2.5 The Eclipse Modelling Framework . . . 24
2.6 Eclipse Epsilon . . . 26
2.7 Logic Programming . . . 28
2.8 Parser generators . . . 32
2.9 Conclusion . . . 34

3 The Probabilistic Vulnerability Analysis model 35
3.1 Introduction . . . 35
3.2 Probabilistic Vulnerability Analysis model design . . . 36
3.3 Probabilistic Vulnerability Analysis Instance model design . . . 41
3.4 The ProbLog model . . . 44

4 Transforming P^{2}CySeMoL models 46
4.1 Introduction . . . 46
4.2 iEaat file format . . . 47
4.3 Deriving PVA models from PRM files . . . 47
4.4 Deriving PVAI models from EOM files . . . 50
4.5 Transformations to ProbLog models . . . 51

5 Implementation 53
5.1 Introduction . . . 53
5.2 Used Software . . . 53
5.3 The iEaat Parser . . . 55
5.4 The Analysis Generator . . . 66

6 Measurements and Evaluation 77
6.1 Introduction . . . 77
6.2 CySeMoL sampling parameters . . . 78
6.3 Analysis execution time measurements . . . 79
6.4 Analysis correctness . . . 82

6.5 Discussion . . . 82

7 Related Work 85

7.1 Attack Graph analysis . . . 85

7.2 Probabilistic Programming . . . 85

7.3 Model transformation for analysis . . . 86

8 Conclusions and Recommendations 87

8.1 Conclusions . . . 87

8.2 Recommendations and Future Work . . . 88

A Metamodels A – 1

A.1 Overview . . . A – 1

A.2 The PVA metamodel . . . A – 1

A.3 The PVAI metamodel . . . A – 5

A.4 The ProbLog metamodel . . . A – 6

B Parser grammars A – 7

B.1 Overview . . . A – 7

B.2 General P^2AMF parser grammar . . . A – 7

B.3 Path parser grammar . . . A – 10

B.4 Derived edge parser grammar . . . A – 11

C Command-line tools A – 12

C.1 Overview . . . A – 12

C.2 iEaat Parser . . . A – 12

C.3 Analysis Generator . . . A – 13

Chapter 1

Introduction

1.1 Motivation

Computer systems are an essential asset of large companies such as financial institutions, utility companies and telecommunication providers. Given the recent developments in smart grids and the Internet of Things, the importance of ICT systems for society is growing as well. People rely heavily on the availability of these computer systems for their day-to-day operations, and disruptions in the availability of these systems can therefore have disastrous consequences. Examples of potential consequences are financial costs, physical injuries, reputation damage, and theft of intellectual property. Due to their critical role for the functioning of society and enterprises, computer systems are under a constant threat of attacks by external parties such as criminals, other companies, and nation states. Therefore, considering the potential consequences, it may come to no surprise that the prevention of these attacks, and the security of computer systems as a whole, is becoming more and more important concerns for enterprises.

From a business perspective, the owners of such critical computer systems undertake security risks assessments to identify the risks involved. Such assessments analyse the state of security of the computer systems within an enterprise, and investigate the potential harm which may be caused by an adversary. Using this information, it is possible to weigh the likelihood of a particular potential attack with the costs involved with the damage caused by that attack.

Approaches to perform these kind of assessments for digital systems have already been developed. Examples of such approaches are the CORAS[28] and OCTAVE methods[1], which provide techniques to investigate the risks and potential harm in a structured manner (see also section 2.2). These methodologies often require security experts to manually perform the assessments, which is becoming a problem due to the increasing size and complexity of the computer systems involved. Therefore, in order to be able to perform the required risk assessments in the future, an automated method for risk assessment is required. Another consequence of the large complexity of the infrastructure involved in such computer systems is a large amount of security events which have to be investigated. These events, which can range from login failures to overheating servers, are collected and have to be analysed by security experts.

At TNO, it has been observed that security experts in the field feel that their work would benefit from the support of automated tools for the purpose of the assessment of threats, vulnerabilities and risks. As an example, security experts require the evaluation of the potential impact of emerging threats to their system. When some parts of their system are vulnerable to a particular threat, immediate action may be required. On the other hand, unnecessary interventions in a computer system may incur costs due to the reduced availability of that system. Therefore, the significance of these threats and vulnerabilities have to be carefully evaluated. Other questions which might require an answer are questions such as "What is the most likely goal of an attacker, given my observations?", and "Which system is the most likely to be compromised?". Automated risk assessment is an approach which is able to provide answers to these questions, by automating the process.

During the collaboration with the KTH university in the SEGRID[89] project, TNO has investigated the use of the Cyber Security Modelling Language (CySeMoL)[35] for the purpose of automating the risk assessment of network infrastructures. CySeMoL is a system which aids in the design of secure enterprise network architectures, by providing an automated risk assessment for the design phase of networked computer systems. CySeMoL has been applied in practice for the design of secure System Control And Data Acquisition (SCADA) systems[35]. The risk assessment provided by CySeMoL consists of an analysis of the potential vulnerability of the computer systems under analysis. The supported systems take the form of network architectures, which are defined using a model of predefined components such as servers, clients and firewalls. Based on these components, CySeMoL is able to provide an indication of the chances of success a potential attacker has to compromise parts of the defined system. In section 2.3, we will go into more detail on CySeMoL and will discuss the details of its analysis algorithm.

1.2 Problem Statement

CySeMoL is aimed at aiding in the development of secure networks during their design. In practice, the need the assessment of security risks is also present for *live* networks, beyond the design phase. In a running network infrastructure, the development of new threats and changes in the (network) infrastructure need to be taken into account. Consequently, TNO is investigating the possibility to apply the risk assessment of CySeMoL to live networks. However, the CySeMoL framework in its current form is not suitable for this type of dynamic analysis. *A major problem in this regard is the amount of time required for its analysis*[35]. Currently an analysis of a large network takes minutes, where depending on the usage scenario, we require the runtime to be a few seconds or less.

Another problem is that of *automating the discovery of a dynamic network topology*. TNO is actively investigating how this can be achieved, which is still ongoing research. However, eventually the results from the automated discovery need to be integrated into CySeMoL in order to execute its analysis on the most recent state of the network infrastructure. In the current design of CySeMoL, its purpose is the modelling and analysing the vulnerabilities of a network under design. Due to this construction, it is currently *not possible to automatically integrate changes to the network topology in existing models*.

1.3 Research Goals

In order to successfully develop a system for automated risk assessment of network infrastructures, we have to tackle the problems stated in section 1.2. To summarize, the modelling and analysis of network infrastructures using CySeMoL is not as flexible as desired. We are unable to alter the models or the analysis. Moreover, we want to investigate whether replacing the current analysis of CySeMoL with a completely different method will result in a faster analysis. To drive this investigation, we have formulated the following research goals. These goals represent prerequisites for the use of CySeMoL in operational environments, and for the development and integration of other analysis methods:

Goal 1 - Improved vulnerability analysis speed The vulnerability analysis of CySeMoL is not fast enough for our goals, the authors show that their testing model with 200 assets took about 2 minutes to analyse. This performance makes effective use in real-time environments impractical or impossible. Therefore, *we aim to improve the speed of the vulnerability analysis of CySeMoL* for large network infrastructures, by an order of magnitude. The intended analysis should provide the same answers as the original analysis performed by CySeMoL.

As a side note, the current analysis of CySeMoL covers the whole network, while sometimes only the probability of a single goal is desired. It might be worthwhile to be able to perform small fast inferences of selected goals.

Goal 2 - Automatable analysis input Our aim is to support the automatic integration of network topology updates. In order to meet our requirements regarding the support of live networks, *it must be possible to automate the input to the vulnerability analysis*.

Goal 3 - Extensible analysis Apart from a changing input to the analysis, itself, we foresee that in the future, new types of attacks will arise. These attacks will need to be supported in order to ensure that the vulnerability analysis remains up-to-date. Moreover, as vulnerability analysis is only a small part of a fully fledged risk analysis, we want to be able to support other types of analysis as well. As an example, the model of the network components supported by CySeMoL could be extended to provide an analysis of the cost of failures within a network infrastructure. Therefore, *we want to produce an extensible and pluggable version of the vulnerability analysis currently provided by CySeMoL*.

1.4 Approach

We have developed two models which can be used to store the information required for performing the vulnerability analysis provided by CySeMoL. In order to construct a extensible and accessible system of models, we use techniques from the field of model-driven engineering. This field focuses on using models and model transformations for the purpose of software engineering in favour of using computer code (for more details see section 2.4).

The first model, the Probabilistic Vulnerability Analysis (PVA) model, stores the definition of network infrastructure components, potential methods for attack, and defences for these attacks. Additionally, the model contains the information required to perform the same vulnerability analysis as defined by CySeMoL. In the second model, the Probabilistic Vulnerability Analysis Instance (PVAI) model, we store a concrete network infrastructure definition. The available components for this definition are based on the components defined in an existing PVA model. The specification of the PVA and PVAI models is discussed in chapter 3.

As CySeMoL's vulnerability analysis is based on years of research on the modelling of attacks and vulnerabilities[37, 35], we would like to integrate this knowledge into our own models. Therefore, we have developed a program which is able to derive all this information from existing CySeMoL models. Using our program, we are able to construct instances of our PVA and PVAI models. Consequently, using this program and our models, it is possible to reconstruct CySeMoL's original vulnerability analysis.

CySeMoL explores all potential sequences of attacks an attacker can execute. By keeping track of the success probabilities of those sequences it is possible to determine those paths which have the highest success probabilities. On an abstract level, this is a probabilistic reachability problem; inferring the probability of reaching a given node in a graph where edges can be traversed with a given probability. An advantage of defining the PVA and PVAI models is that, as a consequence, we obtain the ability to define an alternative analysis for CySeMoL models. In order to leverage this advantage, we have investigated different methods for performing the vulnerability analysis of CySeMoL models, which might improve the speed and scalability of said analysis. In this thesis, we define a method using probabilistic logic (ProbLog)[17], which allows us to specify the in which we model the vulnerability analysis as a probabilistic logic program. In addition, we have automated the construction of these programs by implementing a model transformation for our PVA and PVAI models. By combining the transformation from CySeMoL to our PVA and PVAI models with the transformation to ProbLog, we obtain an automated method for the vulnerability analysis of CySeMoL models using ProbLog.

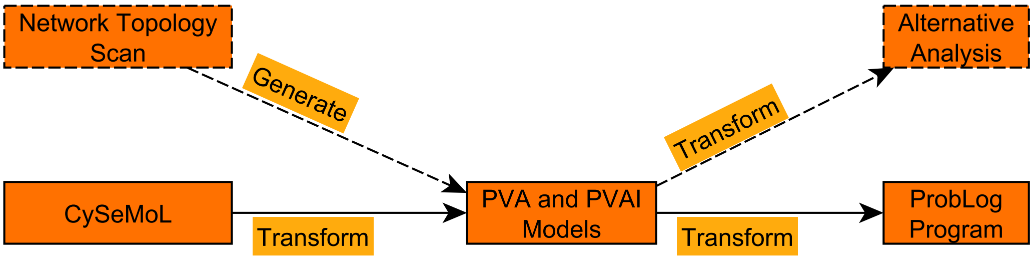


Figure 1.1: An overview of the architecture of models and transformation steps involved in automating the vulnerability analysis of CySeMoL. The dashed objects indicate the potential existence of objects, used to demonstrate potential uses of our approach.

A schematic overview of the required transformation steps and intermediate results is shown in figure 1.1. On the first section of the diagram, we see two potential inputs for our PVA and PVAI models: The original CySeMoL model, and the results from a topology scan of a live network (which takes the form of a PVAI model, based on an existing PVA model). From the obtained PVA and PVAI models, we are able to apply our transformation to ProbLog, but it remains possible to define additional transformations in order to implement another analysis. For the full discussion of design of our developed transformations, we refer to chapter 4 and the more detailed schematic in figure 3.1. The implementation details of the transformations are discussed in chapter 5.

1.5 Validation

Our first goal is to improve the analysis speed of the vulnerability analysis of CySeMoL models. We validate this goal by comparing the execution time of the current analysis of CySeMoL to the execution time of our proposed analysis using ProbLog. The input models for the analysis are specified using CySeMoL and transformed to ProbLog. We scale the size of the models in order to determine the asymptotic behaviour of the analysis times. Furthermore, we validate whether the results of the ProbLog implementation are equivalent to the results from the vulnerability analysis of CySeMoL.

It is difficult to verify our extensibility requirements (goals 2 and 3) within this research, as this depends on how well others are able to use and extend our methods. However, the effort required to repair mistakes in our approach provides information on the extent of its extensibility. Furthermore, we have verified the ability to incorporate changes to the CySeMoL model into our method, as this can provide an indication of the extensibility of our analysis method as well. For a more in-depth discussion of our validation methods and our results, we refer to chapter 6.

1.6 Structure

This report has been structured as follows: We start by providing all the necessary preliminaries required to understand the full extent of our work in chapter 2. In this chapter, we explain the operation and use of CySeMoL, provide details on model driven engineering techniques and some of their implementations, and probabilistic programming.

In the next chapter, chapter 3, we introduce the models which we use to model the network architectures defined using CySeMoL as well as its vulnerability analysis definitions. Chapter 4 discusses how we obtain instances of our previously defined models from models created with the most recent version of CySeMoL (known as P^2CySeMoL). In the same chapter, we explain how we reproduce the analysis of P^2CySeMoL using ProbLog, and how we transform our probabilistic analysis models to a ProbLog program. The implementation details of these transformation processes and the analysis are discussed next in chapter 5. In this chapter, we examine the details of the techniques used to perform the tasks specified in chapter 3.

The validation of our work is described in chapter 6. Within the same chapter, we present measurements of the execution time of our analysis method, followed by an interpretation of the results. In the next chapter, chapter 7, we turn to the work of others which might be related to our work for further reading. We summarize our work and results in chapter 8, and we present ideas which are open for exploration in the future.

We have also included some materials in the appendix to this report, which are useful to reproduce our work. In appendix A we show the metamodels of our PVA, PVAI and ProbLog models. Our model transformation tools need to interpret some of the source code used to define the vulnerability analysis of CySeMoL. For this, we employ the ANTLR4 parser generator, which is able to generate parsers from EBNF grammar definitions (see section 2.8). We list the grammars of our parsers in appendix B. Finally, we provide instructions on the usage of our developed command-line tools in appendix C.

1.7 List of abbreviations

The following table provides an overview of the abbreviations used in this thesis, and their meaning.

Abbreviation	*Meaning*
ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
ATL	ATLAS Transformation Language
CDF	Cumulative (probability) density function
CNF	Conjunctive Normal Form
CySeMoL	Cyber Security Modelling Language
d-DNNF	Deterministic Decomposable Negation Normal Form
EAAT	Enterprise Architecture Analysis Tool
EBNF	Extended Backus-Naur Form
EGL	Epsilon Generative Language
EMC	Epsilon Model Connectivity (layer)
EMF	Eclipse Modelling Framework
EOL	Epsilon Object Language
EOM	Entity Object Model
ETL	Epsilon Transformation Language
JAR	Java Archive
LPAD	Logic Program with Annotated Disjunctions
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Model Group
P^2AMF	Predictive Probabilistic Architecture Modelling Framework
P^2CySeMoL	Predictive Probabilistic Cyber Security Modelling Language
PDF	Probability Density Function
ProbLog	Probabilistic Logic
PRM	Probabilistic Relational Model
PVA	Probabilistic Vulnerability Analysis
PVAI	Probabilistic Vulnerability Analysis Instance
RMSE	Root-Mean-Square Error
SAX	Simple API for XML
SCADA	Supervisory Control And Data Acquisition
SDD	Sentential Decision Diagram
SEGRID	Security for Smart electricity GRIDs
QVT	Query/View/Transform
UML	Unified Modelling Language
UUID	Universal Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language
YAP	Yet Another Prolog

Chapter 2

Background

2.1 Introduction

In this chapter, we will discuss the preliminaries of the concepts and technologies used in this thesis. We begin by providing an introduction to threat modelling and vulnerability analysis in section 2.2. These concepts form the base for the functional rationale of the analysis tool produced for this research. In addition we will briefly investigate some popular methods which are employed for threat modelling.

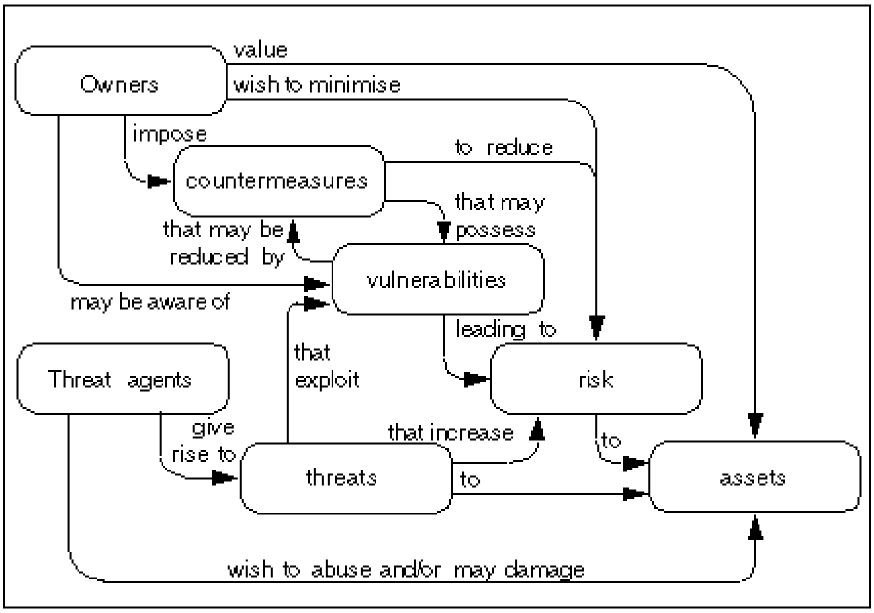
With the security preliminaries in place, we continue by introducing CySeMoL in more detail in section 2.3. Here, we will examine the rationale behind the framework provided by CySeMoL, and how it has evolved over time. We conclude by considering the inner workings of the vulnerability analysis provided by CySeMoL, and provide an example of a vulnerability analysis and its results.

Our aim is to support realistically sized networks, which means that we have to cope with large input models. We require a robust framework for the definition of models which we use the representation of the information required for our probabilistic vulnerability analysis. Additionally, we require flexible methods for the generation and transformation of instances of such models. For these purposes, we use technologies from the field of model driven engineering[72] (MDE), which we introduce in section 2.4. Moreover, we will provide an overview of the relevant MDE frameworks and tools used in this research. Specifically, we discuss the *Eclipse Modelling Framework*, and tools from the *Eclipse Epsilon* project[48].

In an attempt to improve the speed of the vulnerability analysis of network infrastructures, we have replaced the probabilistic analysis of CySeMoL which is based on sampling. Our new analysis is uses probabilistic logic[17] (ProbLog). In section 2.7, we will explain the concept of logic programming, including the preliminaries of logic reasoning. Next, we introduce ProbLog, and describe how it computes marginal probabilities in probabilistic logic programs.

CySeMoL defines P^{2}AMF, a language which is used to define the computation of the vulnerability analysis, which supports the definition of probabilistic computations. We require to automatically dissect and understand programs defined in this language for our transformation purposes. The automated analysis of the structure of a language is known as parsing. We go into detail on this concept in section 2.8. We will introduce the basic concepts of language parsing, and will discuss the tools used in this research which are able to automate this process.

Figure 2.1: An overview of the concepts which arise in the discussion of threat modelling, as defined in ISO/IEC 15408-1:2009[39].



2.2 Threat Modelling

2.2.1 Introduction

The availability of Computer systems and ICT infrastructures is vital for companies. Due to their dependence of these systems, cyberattacks pose a significant threat. Therefore, companies employ security risk assessment in order to obtain insight in the severity and the nature of the security risks that these companies are facing.

We will define some of the concepts which are relevant to the practice of security analysis first. An overview of the terminology is shown in figure 2.1. First, we have to identify the individual system components which are important for a company. These components are known as *assets*, as they are valuable for the company. When we refer to *the attacker*, we denote an individual or group who is potentially trying to attack these assets. If an asset is open to an attack, we say that that asset is vulnerable. The specific way by which some part of the asset is attacked, is known as a *vulnerability* of that asset. Finally, systems and parties which prevent or disrupt an attack are known as *defences* or *countermeasures*.

One of the activities within a risk assessment is the investigation which assets of the company are part of the ICT system, and how these assets are organized. Another follow-up activity is the identification of potential attacks on these assets, and their consequences. An approach to the identification of threats is by creating a model of the threats, which aids in the identification of those threats. This practice is known as ‘threat modelling’. Within threat modelling, multiple approaches which can be taken. For instance, it is possible to model how a system will react, or defend, against attacks. Another modelling approach is the identification of steps which lead to the compromise of a system. In order to assist in the application of threat modelling for complex computer systems, frameworks, tools and methodologies have been developed which aid in the process of manually performing such assessments. We list some examples of such methodologies and tools.

The CORAS method[28] provides a framework for performing risk assessment, it defines visual models similar to UML which aid in the process of the analysis. The CORAS Risk Assessment Platform is a tool based on Eclipse and EMF (see section 2.5), which is used to draw CORAS diagrams. These diagrams are stored using the XMI serialization format (see section 2.5), and consequently provide potential interoperability with external tools.

OCTAVE is another set of methodologies for risk assessment, with goals similar to the CORAS method. The newest OCTAVE-compliant methodology is the OCTAVE Allegro methodology[1], which provides a set of worksheets, which can be used to perform a structured risk assessment of an organization.

Secure Tropos[57] is an agent based modelling language, which allows for a socio-technical analysis. The STS-tool, which is developed by the university of Trento, provides an Eclipse-like environment where Secure Tropos models can be defined. In addition, the tool provides automated analyses in the form of a well-formedness analysis, a security analysis, and a threat analysis.

The Microsoft Security Development Lifecycle (SDL)[65] is aimed to induce a security-aware software development process, and comes with tools to support this. Examples of such tools are:

- The Attack Surface Analyser, which automatically determines the parts of new software which are potentially vulnerable.

- The Microsoft Threat Modelling Tool, which aids in drawing threat model diagrams, and is able to automatically determine potential threats to assets based on the created diagrams.

- The MiniFuzz basic file fuzzing tool, a tool which attempts to find bugs in software by trying out random inputs.

SDL defines seven software development phases, comprising of 'practices' which aim to integrate the design, implementation and verification of software security into the development process. Step 7 of the design phase explicitly specifies the use of threat modelling. The threat modelling is performed according to the STRIDE[38] threat model, which defines six threat categories which aid in determining potential threats to a software application.

Recent research effort has been dedicated to performing automated threat modelling analysis through a menagerie of attack tree formalisms[50]. The analysis provided by CySeMoL is similar to this type of analysis[37]. In the old method of the analysis of CySeMoL models (see section 2.3.2), a network of Bayesian networks is generated, on which the marginal probabilities are estimated using rejection sampling[37]. However, Holm et al. have developed a newer analysis method, which focuses on the modelling of the interactions of the network components. Consequently, its similarity to attack tree analysis has faded to an abstract level; the analysis of structures of attack steps.

2.2.2 Vulnerability analysis

Given a threat model of a system, we are (to the extent provided by the threat model) able to analyse the system for threats. A different type of analysis on a threat model is the identification of vulnerabilities, e.g. the determination of which parts of the modelled system are vulnerable to threats. There are many types of systems for which vulnerability analysis exists. For instance, the SEGRID project aims to develop an analysis of vulnerabilities for smart grids[89].

In this report, we will refer to the aforementioned definition of a 'vulnerability analysis'. Even so, the term is also used for the practice of finding exploitable bugs in software. This can be identified as a subset of our (more general) definition. Still, the practice of finding vulnerabilities in software consists of different concerns, and applies other types of techniques (e.g. fuzzing and symbolic execution).

2.3 The Cyber Security Modelling Language

2.3.1 Introduction

Our work is based on the analysis approach used by CySeMoL, therefore we will explain how the current version of CySeMoL came to be, and define the characteristics of the analysis it provides. The Cyber Security Modelling Language (CySeMoL[37]) was developed at KTH as part of the VIKING[5] project, to aid in the automated analysis of vulnerabilities for the design of SCADA systems. In subsequent research, the analysis method has been improved in a version called P^{2}CySeMoL[35]. This final version of CySeMoL formed the basis of a commercialized version, securiCAD[20], which is under active development by the *foreseeti*[19] company.

2.3.2 Versions

CySeMoL has been under development over time. Our work is based on P^{2}CySeMoL, which is the successor of CySeMoL. The main difference between CySeMoL and P^{2}CySeMoL is the method both tools use to perform their analysis. The old approach CySeMoL, uses a so-called *Probabilistic Relational Model*, which specifies how blocks of Bayesian networks can be generated from system components. The PRM also describes how these blocks relate to each other, which is used to construct a large Bayesian network for the entire system under analysis. The separation between the network component types and the system definition is already present in CySeMoL.

P^{2}CySeMoL and its commercial successor SecuriCAD, use the Predictive Probabilistic Architecture Modelling Framework (P^{2}AMF), to perform the analysis of a defined network. P^{2}AMF provides an alternative method for the inference of the probabilistic model of CySeMoL based on sampling methods. This new approach is able to support larger models while simultaneously providing a faster analysis. P^{2}CySeMoL extends the threat model of CySeMoL by including more types of attacks, defences and assets.

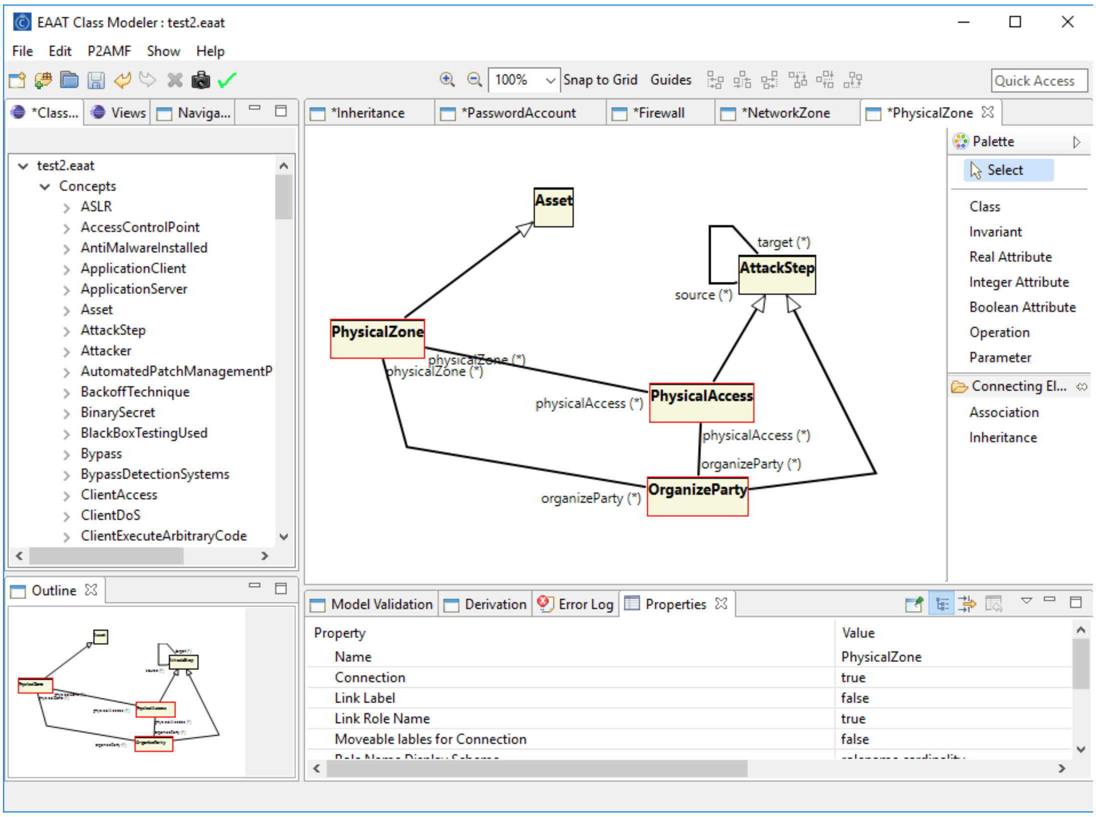
2.3.3 The Enterprise Architecture Analysis tool

CySeMoL models can be created and analysed using the Enterprise Architecture Analysis Tool (EAAT)[7], which consists of two parts. The first part of EAAT is the *Class Modeller*, which is used to graphically define classes with their relations and operations, in a similar fashion as EMF and UML. P^{2}AMF is used to specify the behaviour of the operations, which means that these operations are able to exhibit probabilistic behaviour. The class modeller interface is shown in figure 2.2. The second part of EAAT is the *Object Modeller*, which is used to graphically model and define *instances* of previously created *Class Models*. It provides an interface for the invocation of the P^{2}AMF analysis for the loaded object model. On completion, the results are displayed in the graphical interface. A screenshot of the interface is shown in figure 2.3. The CySeMoL manual[36], contains a full specification of the use cases of the class modeller and the object modeller. It also provides a complete overview of the analysis features, and how these are specified.

The class modeller allows the grouping of classes and their relations into templates. These templates can be used in the object modeller to quickly create multiple instances of the classes defined within the templates. This feature reduces the modelling complexity and allows the composition of functional units of a model within templates.

2.3.4 The CySeMoL threat model

The threat model of CySeMoL specifies how an attacker who is attacking a predefined network infrastructure can be simulated. The components involved in this threat model are defined using the *Class Modeller* of EAAT. We list some components in table 2.1, for the other components, we refer to the CySeMoL papers, and the manual[35, 37, 36]. Within P^{2}CySeMoL, there are four classes which play an important role in its analysis, these classes, which are also shown in figure 2.4, define the main components of a P^{2}CySeMoL analysis.

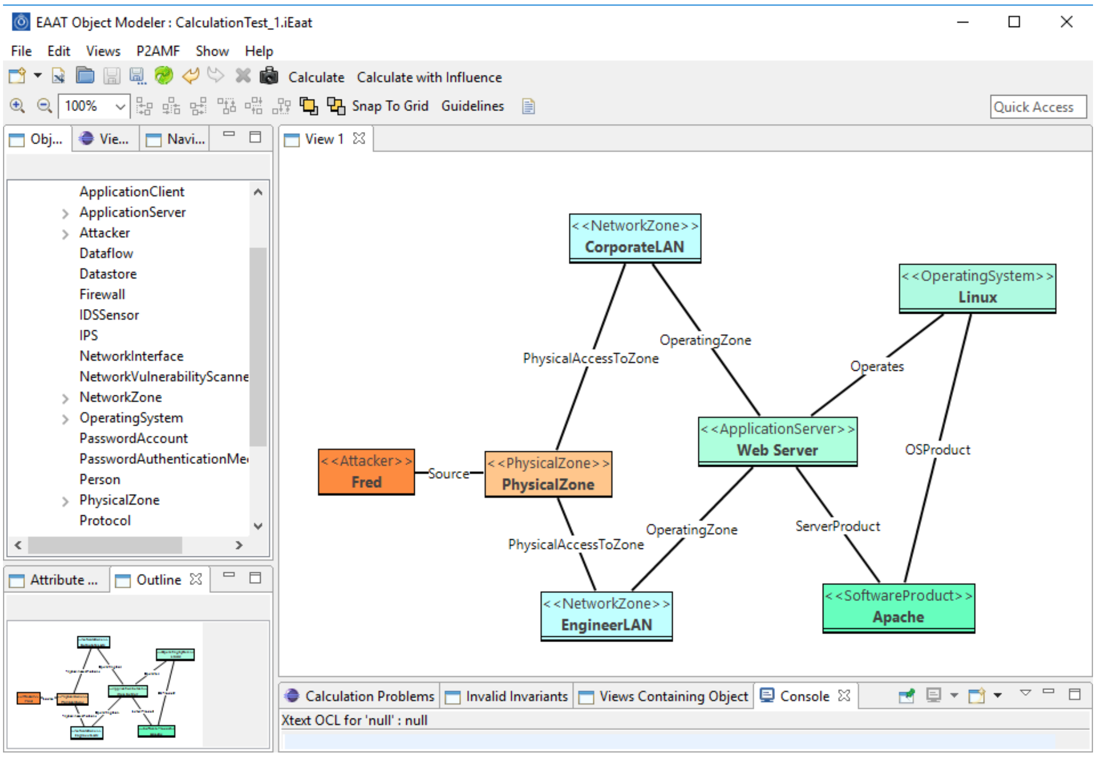


Property	Value
Name	PhysicalZone
Connection	true
Link Label	false
Link Role Name	true
Moveable lables for Connection	false

Figure 2.2: A screenshot of the class modeller, in which a template is being edited.

Table 2.1: Non-exhaustive list of P^{2}CySeMoL assets, and some of their associated attack steps and defences.

Asset	*Attack step*	*Defence*
NetworkZone	FindUnknownEntryPoint ObtainOwnAddress DoS DNSSpoof	PortSecurity DNSSec
PhysicalZone	Access	
OperatingSystem	Access ARPSpoof ExecuteMaliciousPayload FindUnknownService DenialOfService	AntiMalwareInstalled HasAllPatches USBAutoRunDisabled
ApplicationServer	ExecuteArbitraryCode ConnectTo FindExploit	LoadBalancer HasAllPatches



```
Xtext OCL for 'null' : null
```

Figure 2.3: A screenshot of the object modeller, in which a network infrastructure is being defined using templates (indicated by the guillemots). Note the available templates in on the left side of the interface.

The Asset class models components of the network infrastructure which can be attacked. Examples of such assets are servers, clients, and persons. For examples, see table 2.1 and the P^{2}CySeMoL manual[36].

The AttackStep class is used to model a specific attack, or a part of such an attack. The source and target relations are used to link attack steps to assets. An example of an attack step is the ExecuteArbitraryCode attack step, which is linked to the ApplicationServer asset. This attack step models the execution of arbitrary code on this server, and is a prerequisite for the execution of exploits on that server.

The Defence class models countermeasures for attacks. Defences are modelled with a probabilistic isFunctioning method, which allows to specify how the probability that a defence is functioning should be evaluated. For instance, a defence may perform better in the presence of other defences. The existence of these defence classes (which are connected to assets) is used in the derivation code of attack steps to influence the success probability of those attack steps. For instance, if the PortSecurity defense is enabled for the NetworkZone asset, than the ObtainOwnAddress attack step will always fail.

At the bottom of the diagram are the Attacker and AttackStep classes, which model the potential actions the attacker can take. The getPaths() operation of the AttackStep class is used to determine the allowed sequences of attack steps. Whereas the isAccessible() operation is used to determine whether the AttackStep can be performed. The latter operation plays a double role by allowing more fine-grained restrictions on the sequence of attack steps, as well as determining the success likelihood of the attack.

An attacker is allowed to have one or more ‘entry points’. These entry points represent parts of the system where the attacker can begin his attacks. In CySeMoL, entry points are defined through the ‘source’ relation between the Attacker and AttackStep class. As a consequence, an entry point is modelled as an initial attack step which will succeed in all cases.

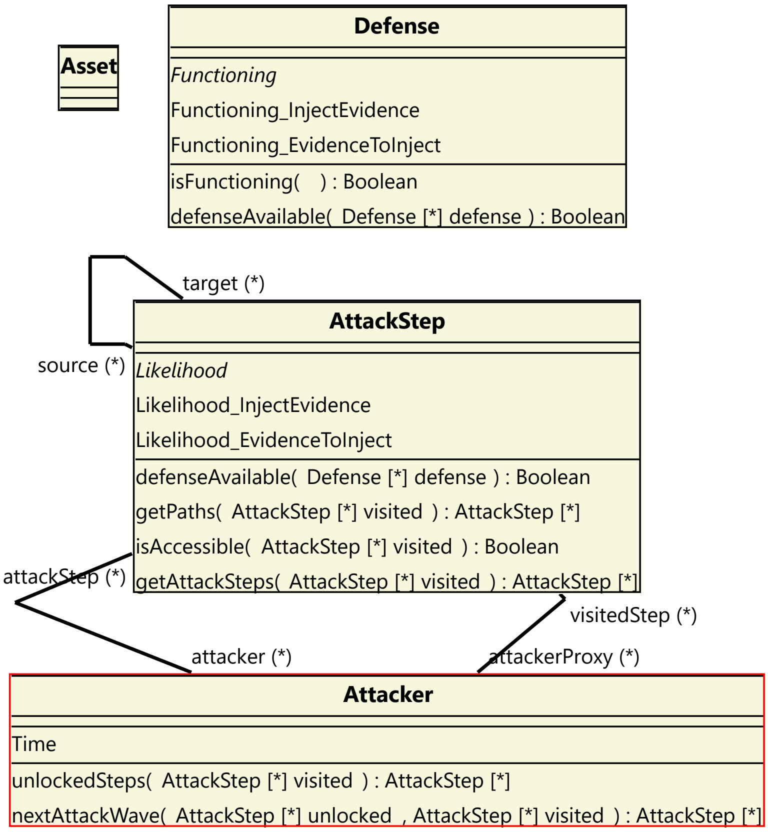

Asset
Defense
Functioning
Functioning_InjectEvidence
Functioning_EvidenceToInject
isFunctioning( ) : Boolean
defenseAvailable(Defense [*] defense) : Boolean
target (*)
source (*)
AttackStep
Likelihood
Likelihood_InjectEvidence
Likelihood_EvidenceToInject
defenseAvailable(Defense [*] defense) : Boolean
getPaths(AttackStep [*] visited) : AttackStep [*]
isAccessible(AttackStep [*] visited) : Boolean
getAttackSteps(AttackStep [*] visited) : AttackStep [*]
attackStep (*)
visitedStep (*)
attacker (*)
attackerProxy (*)
Attacker
Time
unlockedSteps(AttackStep [*] visited) : AttackStep [*]
nextAttackWave(AttackStep [*] unlocked , AttackStep [*] visited) : AttackStep [*]

Figure 2.4: The root classes of P^{2}CySeMoL.

In CySeMoL, the success of an attacker is modelled with respect to the time invested by the attacker. This allows the implementation of the notion that spending more time on an attack makes it more likely that it will succeed. One drawback of the CySeMoL implementation, is that the workdays parameter is defined as a constant which is the same for all attack steps. This design choice allows the analysis of CySeMoL to return a single probability of success, but makes it impossible to estimate the time-to-compromise of an asset.

For individual CySeMoL model instances, EAAT permits the definition of 'evidence'. This evidence amounts to specifying a constant outcomes for any probabilistic attribute in the CySeMoL model instance. When analysing models with evidence, CySeMoL conditions the probability distributions on this evidence. This way, evidence can be used to determine the success probabilities of an attacker, given a set of observations.

2.3.5 Vulnerability Analysis

With the threat model in place, we will now examine how P^{2}CySeMoL calculates the probabilities of success for an attacker. Recall that the method in which P^{2}CySeMoL is defined, assumes a 'workdays' parameter, which indicates the amount of time an attacker spends on each individual attack step. Using this variable as its input, CySeMoL invokes P^{2}AMF to estimate the success probabilities of the attacker.

When starting the analysis, P^{2}CySeMoL first instantiates all probability distributions for the given amount of workdays into a single probability[35]. For example, consider the following instantiation of a cumulative exponential distribution:

$$P(success, w) = 1 - e^{-0.2w} \qquad \lambda = 0.2$$
$$P(success, w = 5) \approx 0.632$$

Next, the model is sampled according to the procedure described in section 2.3.7, however, we will go into more detail on how the OCL program is evaluated. For each sample, each with its own version of the P^{2}AMF code due to the sampling of all probabilistic elements, CySeMoL determines the attack steps which are reachable by the attacker within the context of that sample.

P^{2}CySeMoL searches for reachable attack steps by invoking the recursive `Attacker.nextAttackWave` operation. This operation recursively searches through a graph of attack step sequences. For a given set of reached attack steps $\mathscr{V}$, the operation first determines the frontier set $\mathscr{F}$: the set of attack steps reachable from $\mathscr{V}$, which are not already in $\mathscr{V}$. The reachability relation is implemented through the `AttackStep.getPaths` operation, which returns a set of attack steps reachable from that attack step. For each attack step in $\mathscr{F}$, the `AttackStep.isAccessible` method is evaluated to test whether that attack step is reachable. The accessible attack steps from the frontier set are added to $\mathscr{V}$. More formally, let $\mathscr{A}$ be the set of accessible attack steps, then $\mathscr{V}' = \mathscr{V} \cup (\mathscr{F} \cap \mathscr{A})$.

The `Attacker.nextAttackWave` operation is recursively invoked, until a fixed point $\mathscr{C}$ is reached, in which no new attack steps are reachable. This set $\mathscr{C}$ contains all attack steps which are reachable by the attacker for the sample. Ultimately, the set $\mathscr{C}$ of reachable attack steps is determined for every sample. Next, these sets are aggregated to obtain an overall success probability for each individual attack step in the model. The success probability for a single attack step is determined by the amount of samples in which that attack step was reachable.

2.3.6 Monte Carlo methods

We will introduce the details of how CySeMoL derives the required probabilities in the next section, however, this discussion requires some technical knowledge on Monte Carlo methods. Therefore, we will first describe what Monte Carlo methods are, and how they can be applied for the inference of probabilities.

Monte Carlo methods concern the drawing of many random samples from a system in order to obtain an estimate of its properties. The methods are often used to estimate properties of systems of high complexity[9]. The reason for this, is that by drawing samples, it is possible to quickly obtain a ballpark figure of the probability space. Using exact methods would require an exact inference of this space, which would be too complex. CySeMoL Monte Carlo methods by drawing samples from probability distributions to estimate the likelihood of success for attack steps.

In modern systems, we have access to pseudo-random number generators, which are able to quickly generate large sets of numbers which are (almost) uniformly distributed. These methods use deterministic algorithms (for instance, the Mersenne Twister[55] algorithm), and are therefore not truly random, but suitable for Monte Carlo purposes[55]. In addition to uniformly distributed numbers, we often require to generate numbers which are not uniformly distributed. We will discuss some methods for generating samples from different distributions. The supported distributions of P^{2}CySeMoL are listed in table 2.2.

Table 2.2: The probability distributions used in P^{2}CySeMoL.

Parameters	*Meaning*	*Definition*	*Description*
$c \in [0,1] \subset \mathbb{R}$	Constant	$F(x) = c$	The Bernoulli 'distribution'
$\lambda \in \mathbb{R}$	Rate	$F(x) = 1 - e^{-\lambda x}$	The cumulative Exponential distribution.
$\mu, \sigma \in \mathbb{R}$	Mean, standard deviation	$F(x) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	The Normal probability density function.
$\mu, \sigma \in \mathbb{R}$	Mean, standard deviation	$F(x) = \frac{1}{2} + \frac{1}{2}\mathrm{erf}\left(\frac{\log x - \mu}{\sigma\sqrt{2}}\right)$	The cumulative Log-normal distribution. Where the Gauss error function is denoted by $\mathrm{erf}(x)$.
$\alpha, \beta \in \mathbb{R}_{>0}$	Shape, rate	$F(x) = \frac{\gamma(\alpha,\beta x)}{\Gamma(\alpha)}$	The cumulative Gamma distribution. With $\Gamma(x)$ the Gamma function, and $\gamma(x,y)$ the incomplete gamma function.

Old versions of EAAT employed forward sampling, where every time a probability distribution was encountered, a sample was drawn from the corresponding distribution. An extension of forward sampling, which also supports the sampling of conditional probabilities is rejection sampling[85]. Rejection sampling is executed similar to forward sampling, with the difference that every sample which does not conform to the evidence is discarded. An advantage of this method is that its implementation is simple when forward sampling is already in place. A disadvantage is that in some cases, many samples might be discarded, which requires a large number of samples to be drawn.

With the introduction of P^{2}AMF, EAAT switched its sampling method to the Metropolis-Hastings algorithm[43]. The algorithm as popularized by Hastings[33, 9] is a Markov Chain Monte Carlo (MCMC) method, which generates new samples which depend on earlier samples. During this sampling, the underlying Markov Chain reaches a point where its stationary distribution nears the target distribution. This has the advantage that more samples conform to the desired distribution which reduces the amount of samples which need to be discarded when compared to rejection sampling. A disadvantage of the Metropolis-Hastings algorithm is that, because the underlying Markov Chain needs some time to reach its stationary distribution, some *burn-in* samples need to be drawn. These initial samples do not necessarily conform to the desired distribution, but instead reflect a transient state of the markov chain. Therefore, these burn-in samples need to be discarded.

2.3.7 P^{2}AMF

P^{2}AMF[43], which stands for the 'Predictive Probabilistic Architecture Modelling Framework' is an extension to the OCL language. The extensions include support for uncertainties in model variables and connections between classes. In addition, P^{2}AMF inherits the support for collections, logic formulae and set operations from OCL. The uncertainties are specified using probability distributions, and P^{2}AMF provides a method for performing inference on the resulting probabilistic models[43]. The supported probability distributions are listed in table 2.2. Additionally, P^{2}AMF supports a linear approximation of a probability distributions, by providing it with a list of points from that distribution. These points are used to define a set of linear equations which model the probability density between those points. In section 2.3.5, we will examine a practical application of P^{2}AMF when we explain how P^{2}AMF is used to perform a vulnerability analysis of P^{2}CySeMoL models. An overview of the context of each CySeMoL-related concept which we have discussed is shown in figure 2.5.

An analysis by P^{2}AMF is conducted in the following way[35]: First, a user-specified amount of versions of the model are instantiated. Next, for each version, all probabilistic expressions are sampled and evaluated. After this step, all probabilistic expressions have been cast into regular OCL expressions, which can be evaluated by the OCL parser. Finally, the results from each model instance are aggregated, and presented to the user in the graphical interface.

We will provide an example of the usage of P^{2}AMF. Consider a coin with two sides which can have different (positive) weights for each side. We say that the probability that the coin lands on a side is determined by the fraction of the total weight that side has. Ergo, we can calculate the probability that a coin with two sides which weigh w_{heads} and w_{tails} grams respectively in the following way:

$$P(\text{Heads}) = \frac{w_{\text{heads}}}{w_{\text{heads}} + w_{\text{tails}}}$$
$$P(\text{Tails}) = \frac{w_{\text{tails}}}{w_{\text{heads}} + w_{\text{tails}}}$$

Using P^{2}AMF we can model this problem by defining a `Coin` class with two double-precision attributes `weightHeads` and `weightTails`, which denote the weight of each side. Next, we define a `flipCoin` operation, which will flip this coin and return ‘true’ if the coin landed on the ‘Heads’ side of the coin. The derivation of this operation can be specified as follows, using P^{2}AMF:

```
let probability : Double = weightHeads/(weightHeads+weightTails) in
  bernoulli(probability)
```

This code specifies the operation as a Bernoulli experiment, which results in either true or false, with probability $P(\text{Heads})$. P^{2}AMF can be used to infer the probability of both ‘Heads’ and ‘Tails’. Even though this example is extremely trivial, we note that P^{2}AMF supports the dependency of the weight of the coins on other classes. For example, we could model a minting machine, which outputs coins with different normally distributed weights for each side. This way, the `Coin` class can be reused in more complex models.

2.3.8 Vulnerability analysis example

We will now explain the operation of P^{2}AMF using an example network architecture defined in CySeMoL. Our example concerns the CySeMoL object model shown in figure 2.6. In this model, we model an attacker ‘Fred’, who has successfully broken into a server room, reflected by the `PhysicalAccess` entry point in the `PhysicalZone` instance. In this server room, it is possible to connect to two separate networks, modelled by the two `NetworkZone` template instances. From these networks, it is possible to connect to a web server, which runs Apache on an instance of the Linux operating system. This has been modelled by the `ApplicationServer`, `OperatingSystem` and `SoftwareProduct` instances. All template instances are left at their defaults, except for one network zone, the `EngineerLAN` zone. For this network, we know that its `PortSecurity` defence has been disabled by the administrator. We model this by providing evidence that `EngineerLan.PortSecurity.functioning` is false. We will use the default value for the ‘workdays’ parameter of five work days per attack step.

Returning to our example, we will now examine how the success probability of some example attack steps is determined by P^{2}AMF. We will denote individual attack steps by their type, and will enclose their target in parentheses. For example, the attack step `A` with target `T` will be denoted `A(T)`. In our model, we have specified the `PhysicalAccess` attack step as an entry point for the attacker. This will be our initial visited set $\mathscr{V}_0$. We will list this, and all other relevant sets from our example in table 2.3. Next, the frontier set $\mathscr{F}_0$ is determined by evaluating the `getPaths` operation for every element in $\mathscr{V}_0$.

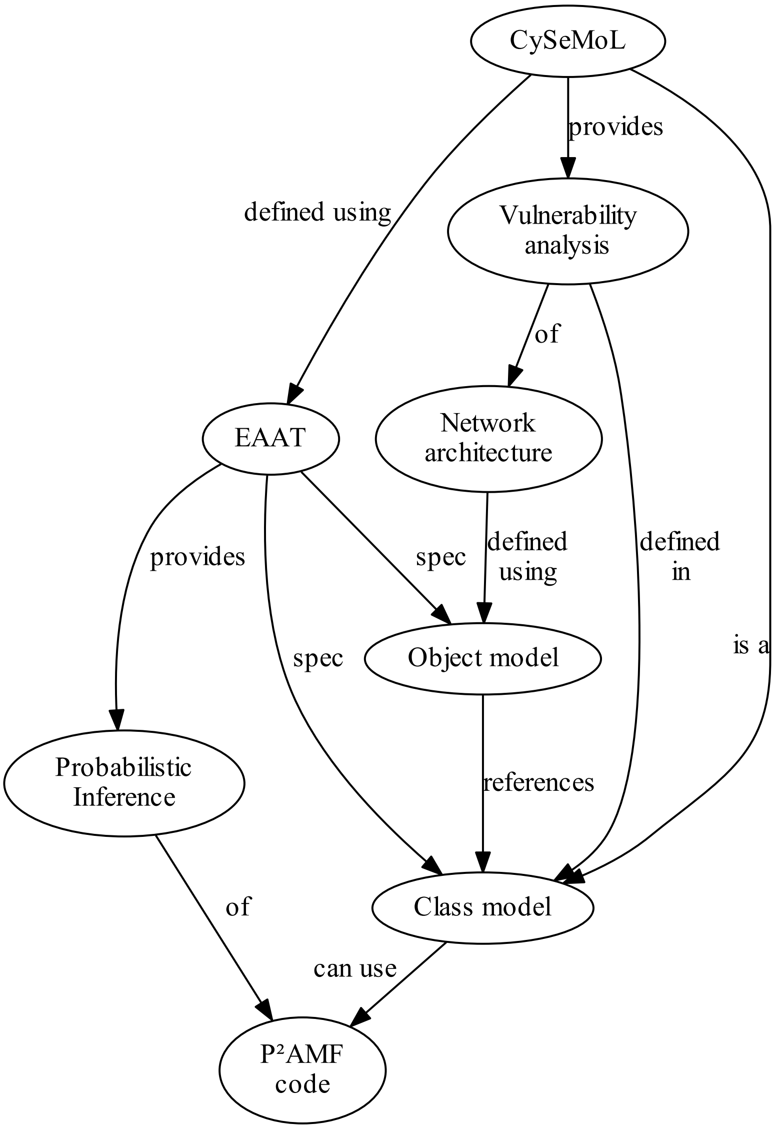


Figure 2.5: An overview of the relations of the relevant CySeMoL-related concepts.

Table 2.3: The progression of the construction of the visited set in our example.

Set	*Contents*
$\mathscr{V}_0$	`PhysicalAccess(PhysicalZone)`
$\mathscr{F}_0$	`OrganizeParty(PhysicalZone)`, `ObtainOwnAddress(EngineerLAN)`, `ObtainOwnAddress(CorporateLAN)`.
$\mathscr{V}_1$	91.6% : `OrganizeParty(PhysicalZone)`, `ObtainOwnAddress(EngineerLAN)`, 50%: `ObtainOwnAddress(CorporateLAN)`.

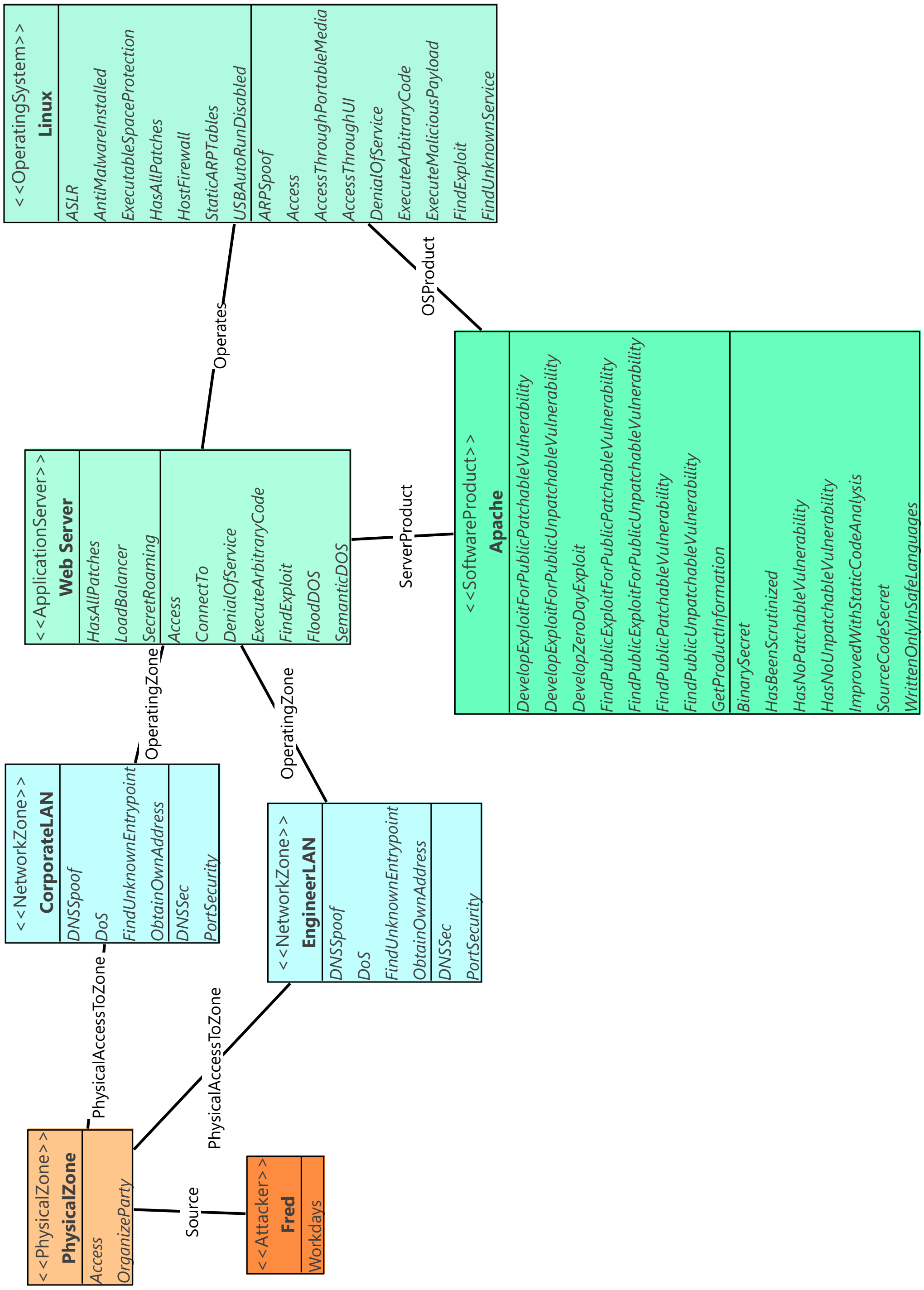


Figure 2.6: A working example of a P^{2}CySeMoL model, as defined using the EAAT object modeller, modelling an attacker and two small networks with a single server.

The first attack step from $\mathscr{F}_0$ is the `OrganiseParty(PhysicalZone)` attack step. This is an attack step we have introduced ourselves for testing purposes, and is not normally part of CySeMoL, it models the probability that an attacker who has access to a physical zone can attempt to organize a party in that area. CySeMoL evaluates the `isAccessible` operation of this attack step, in order to determine whether it should be added to $\mathscr{V}_1$.

In the `isAccessible` operation of the `OrganizeParty` attack step, we encounter a probabilistic distribution. The success probability of this particular attack step has been defined as being cumulative *Gamma* distributed with parameters $\alpha = 0.014593$ and $\beta = 3630.152$. We copied these parameters from a different attack step, and used this resulting distribution to test the ProbLog implementation of the Gamma distribution. Before starting the sampling process, this distribution has been converted to the evaluation of a single probability using the 'workdays' parameter. Since we use the default value of five workdays, the resulting success probability is

$$\begin{aligned} F(x) &= \frac{\gamma(\alpha, \beta x)}{\Gamma(\alpha)} \\ F(x) &= \frac{\gamma(0.014593, 3630.152x)}{\Gamma(0.014593)} && \approx 0.91585 \end{aligned}$$

Analogous to this probability, the `OrganizeParty` attack step will be contained in $\mathscr{V}_1$ in roughly 91.6% of all samples.

The next attack steps are the `ObtainOwnAddress` attack steps for the `CorporateLAN` and `EngineerLAN` network zones. In the P^{2}AMF code of the `isAccessible` operation of this attack step, we discover that this attack step is accessible when:

a) `PhysicalAccess(PhysicalZone)` $\in \mathscr{V}$.

b) The `PortSecurity` defence is not operational.

Item a) is already satisfied at this point. However, we have yet to determine whether the `PortSecurity` defence is operational for each network zone. Recall that we have defined evidence for the `PortSecurity(EngineerLAN)` defence, as being disabled. This satisfies items a) and b) for the `EngineerLAN` network zone, and therefore `ObtainOwnAddress(EngineerLAN)` is added to $\mathscr{V}_1$. The `PortSecurity` defence has no implementation for its `isFunctioning` operation. Instead, the default implementation is used, which states that the defence is functioning in 50% of all cases. Combining items a) and b) for the `CorporateLAN` network zone, we conclude that it will be added to $\mathscr{V}_1$ in approximately 50% of all samples.

The search procedure described above is repeated until no new attack steps are reached. Ultimately, a set of reachable attack steps is obtained for each sample, which is aggregated to estimate the marginal success probabilities for each attack step.

Table 2.4: An overview of the model-driven-engineering languages and techniques discussed in sections 2.5 and 2.6.

Framework	*Modelling*	*General-purpose*	*Model-to-model*	*Model-to-text*
Eclipse Modelling Framework	Ecore	OCL	ATL QVT	Xpand
Eclipse Epsilon		EOL	ETL	EGL

2.4 Model Driven Engineering

2.4.1 Introduction

In order to develop an extensible method for performing the vulnerability analysis of network infrastructures, we have developed models for the representation of these infrastructures, and for the definition of their components and how their vulnerability should be inferred. Such models have the potential to become quite large, for example a telephone carrier must be able to have a network infrastructure model in which each individual phone has to be modelled. Furthermore, we want to support the adaptation of our models, without requiring that our analysis methods have to be redesigned from scratch.

This type of problems also arise for software development, in which complex systems have to be developed which might require to be supported for over 20 years. An example of where the complexity of the long-term support of software is causing problems in practice, is the reliance of some financial institutions on legacy systems written in COBOL[26, 74], a programming language which is over 50 years old. With the need for new methods for the development of complex software, with a focus on the design instead on the implementation details, standards which are based on models rather than code have been developed. A well-known example of such a standard is UML, which is frequently used for the specification of software architectures.

Model Driven Engineering (MDE), is the practice of employing models as a first-class entity in the software engineering process[73], as opposed to computer programs. Due to the emergence of methods for code-generation, and modelling standards (for instance UML[27] and XML[70]), it is currently possible to use models for purposes other than for just the visualization of software design. Techniques from MDE involve the transformation of models into other models and code generation.

For our purposes, we use the MDE techniques of modelling, code generation, model-to-model transformations, and model-to-text transformations. In this section, we will discuss the relevant aspects of these techniques, and will also go into detail on the concrete implementations of the techniques which we use. For this research, we use two frameworks with implementations of the techniques described below. In table 2.4 we provide an overview of the used model driven engineering tasks, and their respective implementations. For an overview of meaning of the abbreviations, we recommend section 1.7.

2.4.2 Modelling

Before we can do anything related to model driven engineering, we have to start with the construction of models. In MDE, the construction of such models is structurally governed by using metamodels. Generally, a model specifies the classes and their relations which are allowed occur in instances of such a model. The notion of what ‘classes’ and ‘relations’ are, is determined through the metamodel. Some common examples of metamodels are: the XML Schema Definition[70], Unified Modelling Language[56], the Meta-Object Facility[60], and Ecore (part of EMF[76]).

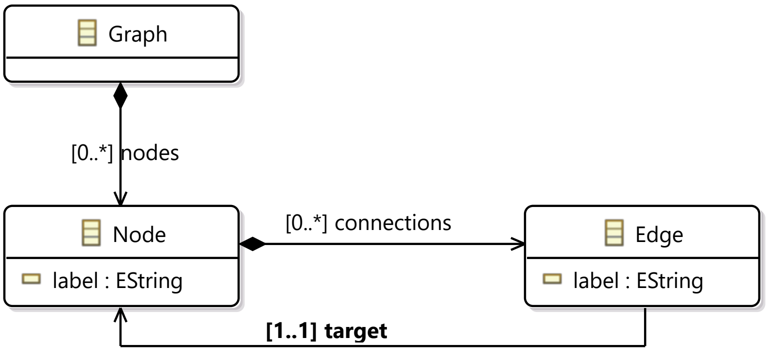


Figure 2.7: An (Ecore) model for representing directed multi-graphs.

For software development applications, developers need only focus on the last two layers of modelling: the model and its instances. Consider the model as shown in figure 2.7, which is a definition (represented using UML) of a model designed for representing directed multi-graphs. This representation is defined using the Ecore metamodel from EMF. Figure 2.8 shows an instance of the model, where the UML inheritance arrows indicate which class the instances correspond to. All instances have their own values for the attributes defined in the model. Finally, figure 2.9 shows what the defined graph would look like. One way to obtain such a visual representation, is to transform the graph model instance to the Graphviz[22] format, and use the `dot` tool to generate the graph image.

2.4.3 Model-to-model transformation

One of the concepts which give rise to practical flexibility of MDE, is the concept of model transformation. Model-to-model transformation is the act of taking one or more input models, and transforming these models to one or more output models in a structured way.

Transformations are often described as being either horizontal or vertical. A horizontal transformation uses the same source and target metamodel. Vertical transformations transform source models to a target model with a different metamodel. Consider a model for a program written in C. Performing a method-inlining optimization can be understood as transforming the C-program to a different C-program. This is an example of a horizontal transformation. Compiling the C-program to assembly code is an example of a vertical transformation.

Model transformations are applied to translate between multiple layers of abstraction[52]. Consider a social network application, where we model users who are connected to other users. We could directly model the connection graph of the users, however, it makes more sense to add one layer of abstraction, and use two models. One model for representing the users, and another for the graphs (for instance, the model we defined in figure 2.7). Using model transformations, we are able to achieve a ‘separation of concerns’, where the users model does not need to take any intricacies of potential target models into account.

Furthermore, models can be reused for different purposes, provided that the models are correctly separated from other models. Instead of generating a Graphviz program for our graphs, we could also generate a database schema, or something totally different. One occasion where this reuse is exploited, is when a *pivot* model is created (see figure 2.10). When using such a pivot model, multiple models are transformed to an intermediate model. This concept follows the same principles as intermediate compilation targets. Apart from providing more flexible support for new transformation targets, using a pivot model also reduces the amount and the complexity of transformations which need to be defined in order to transform between all models. The PVA model (see chapter 3) also has the potential function as a pivot model.

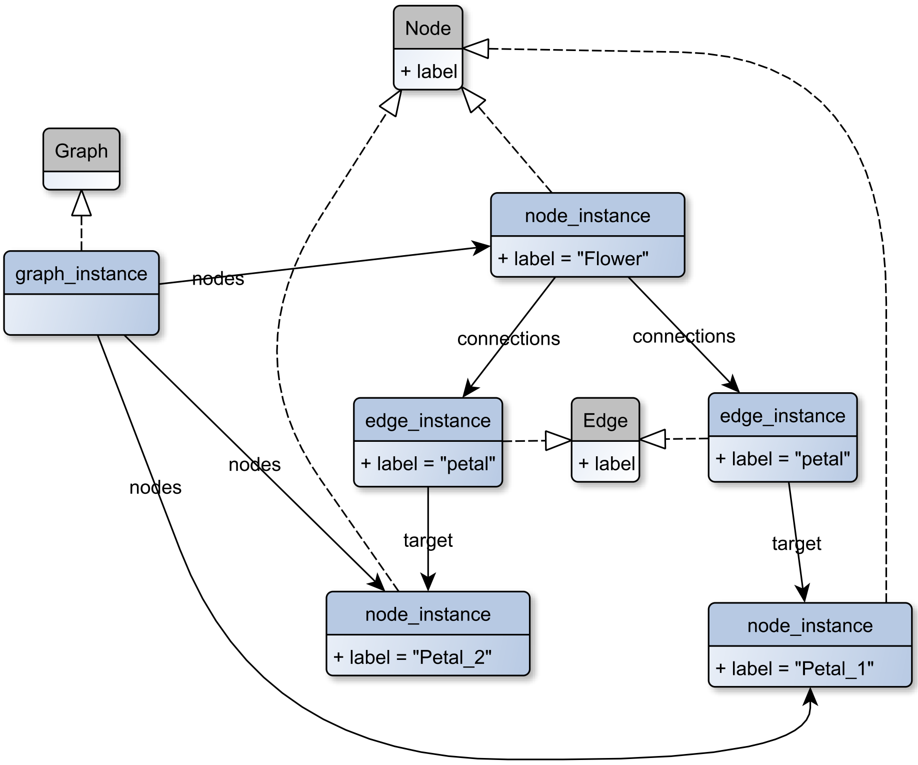


Figure 2.8: An instance of the graph model of figure 2.7, with three nodes, and two edges.

The model transformations themselves are often specified using specialized model transformation languages, such as QVT[52], ATL[44] and ETL[49]. These transformations languages are either 'declarative', 'imperative', or a mixture of the two. In declarative transformation languages, the transformations are specified using rules. These rules in turn specify how to transform some element from the source language to the target language. The order of applying the transformation rules is determined by the transformation engine. Imperative languages explicitly specify the order of transformations, they advocate a more procedural style of programming. In practice, features from both declarative and imperative languages have their own strengths and weaknesses for practical transformations[49]. Therefore, there are also languages which support a mixture of declarative and imperative transformation specifications.

2.4.4 Model-to-text transformation

In addition to model-to-model transformations, a special type of transformation are those which generate text output. From an abstract perspective, we can consider text output as simply another type of model. However, because there are special tools for this class of transformations, we will discuss this kind of transformation in more detail. Furthermore, we have implemented some of our transformations as a model-to-text transformation (see section 4.5.4).

Using model-to-text transformation, it is possible to transform models to programs and other structured text. Examples of potential target formats are XML, JSON, HTML, C, Java, LaTeX, SVG, Graphviz, or any other text-based format. From the perspective of software engineering, code-generation is one of the most compelling applications of model-to-text transformations[73].

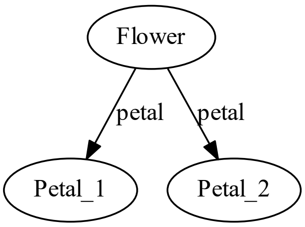


Figure 2.9: The graph as defined by figure 2.8.

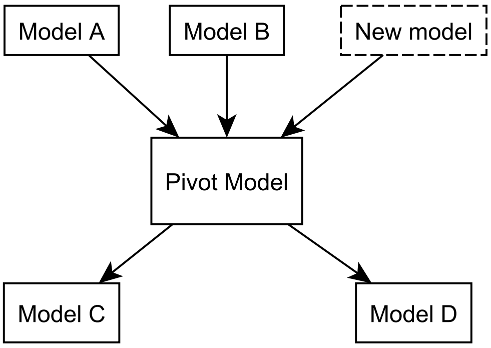


Figure 2.10: Demonstration of a *pivot* model. Instead of having to create new transformations for the new model, only one new transformation needs to be defined. The gains rapidly increase when more models are added.

The concrete model-to-text transformation is defined using *templates*. Examples of some common template languages are Xpand[18] and the Epsilon Generation Language (EGL)[69]. These languages contain specialized constructs to separate text generation from procedural code. For instance, Xpand treats any tokens not inside guillemot signs ('«' and '»') as text which is to be generated. Another common features of template languages are aggregate operations over model types, which allow the definition of operations for all objects of some type. This has the potential to reduce the amount of code required to specify text generation over types which occur in different contexts. For our work, we have used EGL, which we will discuss in more detail in section 2.6. Its application for our work is discussed in sections 4.5.2 and 4.5.3.

Another concept which is relevant in the context of model-to-text transformation is its reverse, where models are defined using *Domain Specific Languages* (DSLs). These languages are designed to describe some model, and usually come with parsing technology (Xtext[24] is a popular framework for this purpose) which is able to automatically generate the corresponding model elements, based on a program written in a DSL.

```
context Person
inv validAge : self.age >= 18
```

Listing 2.1: An example of a simple OCL constraint.

```
context Person::befriend(stranger : Person) : Void
pre : not self.nemeses->contains(stranger)
post : self.buddies->contains(stranger)
```

Listing 2.2: An example definition of a method, with pre- and postconditions. In the example, we define a `befriend` method, which takes another person as an argument. The operation may only be executed when the argument is not in the list of nemeses (we assume that such a list exists) of the current context. When the operation completes, we require that the buddies list contains the person which was passed as the argument to the operation.

2.5 The Eclipse Modelling Framework

2.5.1 Introduction

The Eclipse Modelling Framework is a popular collection of open-source MDE tools[76]. It provides facilities for the definition and execution of models, model transformations, (custom) domain specific languages, and code generation.

The central part for modelling using EMF is the *Ecore* metamodel. It is defined in terms of itself, and is used as the metamodel for all EMF other models. It is inspired by the MOF (Meta-Object Facility) standard from the Object Management Group. To integrate with arbitrary EMF facilities and tools, it suffices to support the Ecore metamodel. EMF provides two built-in model editors for Ecore models, based on Eclipse. The first editor, the *Ecore model editor*, provides a view of the tree structure of the model elements, and supports the specification of all Ecore features, according to the Ecore metamodel. The second editor, the *diagram editor*, features a UML-like interface for specifying views for Ecore models. In addition, it supports the creation of most features from Ecore models. Diagrams are stored in `aird` files, which are coupled with the Ecore model. This separate coupling (as opposed to storing the diagrams in the Ecore file, if that would be possible) enables multiple diagrams to be created to allow for multiple different views of Ecore models.

2.5.2 Model invariants

In some cases, we want to restrict the amount of valid models, based on the values stored within those models. For instance, if we have a class `Person` who has a field `Age`, we might want to require that its value is always greater than zero. This is an example of a model which contains invariants. For the specification of invariants on models, EMF provides the Object Constraint Language (OCL). Using the OCLinEcore technology, these constraints can be embedded into Ecore models. OCL constraints are specified for a context, usually an instance of a class. The aforementioned invariant can be specified as shown in listing 2.1.

Apart from invariants, OCL also supports the specification of pre- and postconditions on operations. These pre- and postconditions are used to specify the state of the model before and after the execution of operations. In listing 2.2, we have defined an example operation with a pre- and a postcondition. Furthermore, the language provides a set-operations and a standard library with auxiliary functions to enhance the modelling capabilities of the language.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <nl.utwente.fmt.pvai:PVAIContainer
3   xmi:version="2.0"
4   xmlns:xmi="http://www.omg.org/XMI"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xmlns:nl.utwente.fmt.pvai="http://nl/utwente/fmt/pvai">
7     <attacker name="Fred"></attacker>
8 </nl.utwente.fmt.pvai:PVAIContainer>
```

Listing 2.3: An example of a XMI serialization of a PVAI model, containing a single instance of the Attacker class, for which the name field has been set to 'Fred'.

2.5.3 Model transformation languages

For model transformation purposes, EMF provides facilities for the specification and execution of model-to-model transformations for the QVT and ATL languages. QVT is a set of model-to-model transformation languages aimed at transforming MOF models into other MOF models. The difference between imperative and declarative transformations as mentioned in section 2.4.3 is also observed in the QVT languages. QVT provides a unidirectional imperative transformation language, as well as a bidirectional declarative language.

ATL is a declarative model transformation language, which supports the transformation of a set of input models to a set of output models. Its programs define transformation rules, where the transformation engine determines a suitable transformation order for objects. The language distinguishes non-essential operations from transformation operations through the support for 'helper' functions. These functions provide a better separation between transformation code and calculations.

2.5.4 Serialization

XMI, which stands for XML Metadata Interchange[31], is a serialization format for MOF models based on XML. It is commonly used as a encapsulation vehicle for passing models between transformation steps in transformation workflows, and for storing model instances on a filesystem. Being based on XML, it is also used for transferring model data using web services[23]. EMF contains an implementation of XMI for the serialization and deserialization of Ecore models.

An example of a XMI serialization of our PVAI model (see section 3.3) is shown in listing 2.3. In the first line of this example, we have the XML identification line which provides an indication that the XML format is used, and contains some metadata which might be relevant to XML parsers. The next tag denotes the existence of an instance of the `PVAIContainer` class. Lines 3 until 5 provide some metadata which describe the version and schema of the XMI implementation which was used. In line 6, the namespace of the metamodel which is used is stored, which is used by the XMI reader to reconstruct the model objects stored in an XMI file. Nested within the `PVAIContainer` tag is an `attacker` tag, which specifies an instance which is contained in the `PVAIContainer.attacker` relation. As there can only be one attacker instance in this relation (see also section 3.3 and appendix A.3), the class of the attacker is not specified. The `name` field of the attacker is specified using the `name` attribute of the `attacker` tag, and is specified as 'Fred'. In the last line, the `PVAIContainer` tag is closed, which concludes the container definition and the XMI file.

2.6 Eclipse Epsilon

2.6.1 Introduction

In addition to the languages provided by EMF, Eclipse Epsilon is another framework for MDE tasks. For our model-driven engineering goals, we will use the technologies and languages provided by this framework. The goal of Eclipse Epsilon is to provide accessible and flexible task-oriented languages for a wide range of model-driven engineering purposes.

The framework is centred around a newly defined general purpose language, the Epsilon Object Language (EOL)[47, 48]. Using this language, it is possible to create, read and modify models. The Eclipse Epsilon framework provides an execution engine for EOL, which runs on the Java Virtual Machine. The Epsilon Object Language is designed in such a way that most of the OCL syntax is also valid EOL syntax.

Eclipse Epsilon is able to interact with multiple model technologies, among which EMF and XML. This is implemented through the Epsilon Model Connectivity (EMC) layer[47]. This layer provides an abstraction to hierarchical models, and provides general operations such as the addition and removal of classes, and the definition of inheritance. Models are loaded in read-only, write-only or read/write mode, after which all other Epsilon tools are able to refer to the loaded models. The Epsilon framework provides EMC implementations for models and metamodels stored using the Ecore, XML and XMI formats. These implementations enable the tools from the Epsilon framework to be used on EMF models.

2.6.2 The Epsilon Object Language

The Epsilon Object Language (EOL) is general-purpose programming language, which is used as the base for all other languages from the Epsilon family. An important feature provided by EOL is the ability to *import* code from other scripts, which greatly enhances the portability and reusability of EOL scripts. EOL supports the iteration constructs often found in imperative languages, specifically while loops, if-then-else and switch constructs, and collection iteration. Similar to the helper functions from ATL, EOL introduces user-defined operations, which may be invoked in other code. In order to quickly optimize expensive operations, EOL has a caching annotation, which causes the engine to cache values for that operation.

The execution engine of EOL uses a *dynamic dispatch* mechanism[47]. This means that when an overloaded method is called, the concrete method which will be invoked is determined during runtime, based on the runtime types of its arguments. A similar dispatch mechanism is used by the Groovy[77] and JavaScript languages, whereas Java implements a limited form of dispatch using virtual methods. Nevertheless, EOL is able to import and use *native Java* classes when they are available on the classpath of the execution engine. Other execution related features of EOL are the support for exceptions and transactions. This allows scripts to gracefully handle potential errors during execution.

Two very useful features introduced by EOL are the support for *extension functions and properties*[47]. An extension function is a new function which is defined from the scope of an existing class. Such functions may refer to the methods and variables available within the public scope of such classes, as if they were defined within the scope of the class itself. Extension functions are useful for transformation purposes, as it allows the encapsulation of frequently encountered operations for a single type. This allows EOL scripts to define additional functionality for objects without changing their source code (which might not always available). An added advantage is that operations which are only relevant from the context of the scripts are only used within those scripts, and do not require additional code elsewhere. Extension properties are similar to extension functions, but add a new field to existing class instances. This is useful to store intermediate results or auxiliary information in existing model objects without having to make changes to the metamodel.

2.6.3 The Epsilon Transformation Language

We will now dive deeper into the task oriented languages themselves. The first task-oriented language which we will discuss is the ETL language, the Epsilon Transformation Language, which is oriented at the specification and execution of model-to-model transformations. The ETL language extends EOL by introducing additional language features on top of the specification described in section 2.6.2. Being a part of the Epsilon ecosystem, ETL is able to use the Epsilon Model Connectivity Layer to support transformations from multiple input models to multiple output models[47].

From a programming perspective, the language is designed as a 'hybrid' transformation language by supporting features from both the imperative and declarative styles of model transformation programming. ETL introduces transformation rules which can be invoked to transform a source entity to a number of target entities. On the other side of the spectrum, ETL allows to define a complete model-to-model transformation using imperative EOL operations.

As mentioned, ETL introduces the concept of transformation rules. These language construct define a (unidirectional) mapping from a single instance from one of the source models to one or more instances from the target models. When the ETL execution engine is invoked on an ETL program, normal rules are applied to all instances of the source type. The result of which is the generation of new instances for the target models. Instead of transforming all instances, it is possible to filter the selection of source instances by defining guards for the transformation rules. These guards are defined as boolean functions over the input instance, which determines whether that instance is applicable for transformation by the rule for which the guard is defined.

It is also possible to obtain the resulting objects from the transformations during the transformation by invoking the special `equivalent` or `equivalents` functions on instances from the source model. Whenever possible, the ETL engine will obtain the transformed instance for the source instance, which can then be accessed from the code which requested the transformation. Related to this feature is that ETL supports so-called 'lazy' rules (a concept borrowed from ATL[45, 47]), which are transformations which are not applied automatically, but only so when they are explicitly requested by invocation of one of the `equivalents` functions.

Finally, ETL supports inheritance of rules to facilitate the transformation of hierarchical class structures. A child rule which extends another parent rule first applies the transformation defined in the parent rule, after which the child rule is executed, which obtains access to the transformation result from the parent rule.

2.6.4 The Epsilon Generation Language

Another language from the Epsilon family which we use in our research is the Epsilon Generation Language, or EGL. This language, which extends the EOL language (see section 2.6.2), is oriented at model-to-text transformation, and is a template-based code generator[47]. Programs in EGL use special separator tags ('`[%`' and '`%]`') to distinguish between the dynamic and static parts of the program. The static parts are directly exported to the output of the program, whereas the dynamic parts are evaluated, and are able to alter their text output.

Within the dynamic parts, it is possible to use any features from the EOL language, including the definition of new functions. The dynamic code has access to the special `out` parameter, which has methods which will export text to the output. Instead of using the `out` parameter, it is also possible to use the special `[%=expr %]` construct, which will evaluate `expr` and directly append its result to the output.

EGL introduces the `TemplateFactory` class, which allows EGL scripts to load and manipulate other scripts (called templates in this context). It is possible to add or change variables to other templates, to procedurally execute their code and subsequently obtain the generated text. This feature improves the modularisation and reuse of templates by allowing the development of dynamically configurable general-purpose templates. These general-purpose templates can then be reused within other templates.

Table 2.5: Truth table for some commonly used logic functions over boolean variables.

x	y	$\neg x$	$x \land y$	$x \lor y$	$x \rightarrow y$	$x \leftarrow y$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$

2.7 Logic Programming

2.7.1 Propositional Logic

In this thesis we often refer to logic programming, therefore it makes sense to introduce the basics of logic programming. However, we will begin by considering its theoretical origin: predicate logic and propositional logic.

Propositional logic is a branch of logic, which is used for reasoning about propositions (hence the name). These propositions take a truth value, they are either $\top$ ('true') or $\bot$ ('false'). Some examples of a proposition are:

"It is currently raining"
"The TV is switched on"
"The house is empty"

Over these propositions, we define logic functions, which are mappings $f : \mathbb{B}^n \rightarrow \mathbb{B}$ of input variables to a single truth value. In this thesis, we will refer to some of these logic functions, which are listed in table 2.5. Using these functions, we are able to compose new complex formulae, which allow us to specify more complex relations and facts about propositions. For example, consider the following formula:

$$(R \land T) \rightarrow (\neg E)$$

We choose to interpret this formula as follows: R states that it is raining, T that the TV is switched on, and E that the house is empty. This formula then specifies the proposition that if it is raining, and the TV is switched on, then the house is not empty. Note that it is not of importance for the construction of formulae whether they make sense, as this is dependent on the interpretation of the formula.

2.7.2 Predicate Logic

Predicate logic (also known as first-order logic) extends propositional logic by adding quantification over individuals to logic formula. Using quantification, we are able to specify statements over a range of values from some domain. We use the for-all ($\forall$) quantifier to quantify over all possible values of the quantification domain, and the existence ($\exists$) quantifier to specify that at least element from the domain satisfies the statement. We list the following examples of formulae and an example interpretation:

$$\begin{array}{rr}
\forall_{x,y,z}\big[(\mathtt{gt}(x,y) \land \mathtt{gt}(y,z) \rightarrow \mathtt{gt}(x,z)\big] & x,y,z \in \mathbb{R}, \mathtt{gt/2} : \;>, \quad > \text{ is transitive} \\
\exists_x\,[x \neq 0] & x,0 \in \mathbb{R}, \quad \mathbb{R} \text{ contains values other than } 0 \\
\forall_x\big[\exists_y[x > 0 \rightarrow x + y < 0]\big] & x,y \in \mathbb{R}, \quad \text{Mixed quantification example}
\end{array}$$

Moreover, predicate logic introduces the concept of predicates. These are named groups of atoms, variables, and other predicates. Their use includes the specification of relations between other elements of the formula. For instance, the `gt` predicate in the example specifies a binary relation between two variables. In the example, we interpret it as the 'greater than' relation for real numbers. Predicates are identified by their name and the amount of arguments they require, the latter amount is known as the 'arity' of a predicate, and is denoted using a forward slash notation. As the `gt` predicate from our example carries two arguments, we refer to it as `gt/2`.

The statements of these examples all hold for their respective domains, however, we are free to specify false statements. Finding a satisfying interpretation for an arbitrary formula in general first-order logic is undecidable (it is possible to express Peano arithmetic in first-order logic)[29, 81].

2.7.3 Horn Clauses

Horn clauses are defined as a subset of predicate logic[79]. A Horn clause is a universally quantified disjunction of clauses, in which at most one clause can be positive, whereas the other clauses must be negated. For instance, we can have the following Horn clause:

$$\forall X\,[\neg cat(X) \vee animal(X)]$$

A common notation for Horn clauses uses a conjunction of clauses and inverse implication. The implication-notation for the previous example is:

$$\forall X\,[animal(X) \leftarrow cat(X)]$$

which can be interpreted as "If something is a cat, it must be an animal". As Horn clauses are defined as always being universally quantified over all variables, the forall quantifier is usually left out.

2.7.4 Logic Programs with Annotated Disjunctions

One method for integrating probabilistic components into Horn Clauses are Logic Programs with Annotated Disjunctions[82] (LPADs). Vennekens et al. demonstrate[82] that LPADs can be used to express Bayesian Networks, and other probabilistic models. LPADs extend logic programming by adding (disjunctive) probabilities to the left-hand side of rules. Such rules are of the following form (as defined by [82]):

$$(h_1 : \alpha_1) \vee \ldots \vee (h_n : \alpha_n) \leftarrow b_1, \ldots, b_m.$$

Here, α_i defines the probability that atom or literal h_i holds, given that the right-hand side of the rule can be proven. To maintain correctness, α_i must be in the range $[0,1) \subset \mathbb{R}$. As an example, we can express the tossing of a coin using LPADs in the following way:

$$\begin{aligned}(heads(coin) : 0.5) \vee (tails(coin) : 0.5)\\ \leftarrow toss(coin).\end{aligned}$$

2.7.5 ProbLog

ProbLog[14] builds on the ProLog logic programming language[11], which is a declarative language with a focus on logic reasoning. Programs defined in ProLog are analogous to a set of *Horn clauses*. Being based on ProLog, ProbLog inherits most of its syntax elements, including lists and tuples[15]. The most notable non-supported operations are the cut-operator (`!/0`) and the if-then-else rule (`if/3`). Instead of just $\top$ and $\bot$, ProbLog labels each fact with the probability that it is true[14]. By supporting this syntax, ProbLog introduces support for LPADs, which is demonstrated by the following ProbLog program:

```
0.5::heads1.
0.6::heads2.
twoHeads :- heads1, heads2.
query(heads1). % 0.5
query(heads2). % 0.6
query(twoHeads). % 0.3
```

The predicate `0.5::heads1` indicates that if this predicate were sampled, it would be found to be $\top$ in 50% of all cases. By consisting of a conjunction, the `twoHeads` predicate can only be proven $\top$ when the both the predicates `heads1` and `heads2` are proven $\top$. The `query/1` predicate instructs ProbLog to infer the probability of its argument.

Another feature of ProbLog is the support for 'intensional probabilistic facts'. These facts are of the following form:

```
0.8::hearsAlarm(X) :- person(X).
person(john).
query(hearsAlarm(_)).
```

During analysis, the variable `X` in this example is substituted with all suitable ground facts. In this example, there is only a single ground fact, namely `person(john)`. The resulting ground program is as follows:

```
0.8::hearsAlarm(john) :- person(john).
person(john).
query(hearsAlarm(john)). % 0.8
```

These intensional probabilistic facts are useful for specifying multiple probabilistic rules at once.

In addition to constant probability annotations, ProbLog also supports variable probability annotations, provided that they can be evaluated at runtime. We use this feature to implement probability distributions. For instance, consider the following program:

```
P::expCDF(X, Lambda) :- P is 1 - exp(-Lambda * X).
query(expCDF(0.1,8)). % 0.550671
```

ProbLog provides auxiliary functions for floating point calculations, of which the `exp/1` function (which implements $f(x) = e^x$) used above is an example.

2.7.6 Probabilistic inference of ProbLog programs

The latest engine of ProbLog (ProbLog2) is implemented in Python, and therefore runs on all systems which have support for the Python language. ProbLog supports multiple knowledge compilation tools, among which Sentential Decision Diagrams[46], and two d-DNNF (Deterministic Decomposable Negation Normal Form) compilers DSHARP[58] and C2D[13]. Inference using Sentential Decision Diagrams (SDDs) is the most recent method for inferring marginal probabilities in ProbLog programs. We will briefly describe the how ProbLog uses SDDs for inference of probabilistic logic programs, for the technical details, we refer to the Fierens et al.[25].

First, the ProbLog program is grounded, which means that all intensional facts are instantiated, and that as many variables are instantiated to predicates which are true in their context. This process is similar to finding the Least Herbrand Model[25]. During grounding, ProbLog reduces the program by removing rules, evidence and facts which are not required to prove the queries in the program. The resulting program is usually smaller, which saves time in the following analysis steps.

When the ground program has been constructed, it is converted to CNF, in such a way that the closed-world assumption of ProLog programs is retained. This is achieved by applying Clark's completion[10, 41], along with cycle-breaking for recursive rules[25]. Next, any evidence is included into the formula by constructing the conjunction of the formula for the evidence and the program formula.

The resulting logic formula is converted to a weighted formula, by assigning weights to each truth value of each proposition with an annotation in the ground program. The value of each weight is equal to the probability as specified in the ground program. After this step, the problem of inferring the marginal probabilities has been transformed to an instance of the WMC (Weighted Model Counting[30]) problem.

The weighted model counting problem is solved by compiling the formula into a sentential decision diagram using the approach described by Vlasselaer et al. [83, 84]. In the same work, the authors define a method for efficiently solving the WMC problem when the SDDs have been obtained. The solution to the WMC problem is interpreted within the context of the ProbLog program, and the marginal probabilities are presented to the user.

2.7.7 Additional ProbLog features

In addition to the normal mode in which the marginal probabilities of a probabilistic logic program are inferred, ProbLog supports different operational modes. We will briefly two modes which overlap the subject matter of this thesis, the ’shell’ mode and the ’most probable explanation’ modes.

Shell mode ProbLog features a ‘shell’ mode, where users are able to load ProbLog programs, and then execute queries against these programs. This mode of operation is more similar to how ProLog implementations usually behave. The ‘shell’ feature uses the shell of the operating system, and can therefore be used by external software to interactively execute queries.

Most probable explanation Another feature of ProbLog is its ‘most probable explanation mode’ (MPE). For this mode, ProbLog uses the concept of ’possible worlds’, which form the set of assignments of the probabilistic clauses of a given program. The MPE mode determines the possible world, where all evidence and queries are true, with the highest probability.

2.8 Parser generators

2.8.1 Introduction

In order to transform the P^{2}AMF programs used by CySeMoL, we require a method for systematically analysing and storing such programs. Building a data structure for the representation of languages is known as 'parsing'. In this section, we will discuss the theory and practice regarding parsing. We will first introduce the notion of formal languages, and grammars over languages.

A (formal) language L over an alphabet Σ, is defined as a set of strings of words Σ^* from that alphabet, i.e. $L \subseteq P(\Sigma^*)$. When we put constrains on the structure of words which is allowed to occur in languages, we obtain classes of languages which satisfy these constraints.

One method for specifying structural constraints on a language are (formal) grammars. Grammars define how valid sentences of a formal language can be generated by applying transformation rules, starting from a starting symbol S. For example, we can define the language over the alphabet $\{a\}$ of sequences of 'a' symbols with arbitrary length as follows:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow Sa \end{aligned}$$

where ϵ stands for the 'empty string', the string containing no symbols at all. By starting from the start symbol S, we can form any sequence of 'a' symbols by applying the two rules. For instance, the following derivation proves that the string 'aaa' is in our language:

$$\begin{array}{rr} S & S \rightarrow Sa \\ Sa & S \rightarrow Sa \\ Saa & S \rightarrow Sa \\ Saaa & S \rightarrow \epsilon \\ aaa & \end{array}$$

Note that the second rule is a recursive definition, the symbols occurring on the left-hand side of the rule, also appear on the right-hand side.

In this example, we have used the two types of symbols which occur in grammars: *terminal* (a) and *non-terminal* (S) symbols. Non-terminal symbols are not part of the alphabet of the language, and serve as intermediate symbols used to structure the application of rules. Terminal symbols on the other hand are part of the language alphabet, and will eventually form the sentences generated by the grammar.

Similar to the restrictions we can impose on languages, we can create classes for grammars, by restricting the form of the transformation rules. These restrictions have given rise to the Chomsky-hierarchy, which divide all grammars into four categories: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars. For the specification of parsers, we often use grammars from the family of context-free grammars or CFGs. CFGs are defined as grammars which only have a single non-terminal symbol at the left-hand side of their rules. This means that the application choices of rules depend on non-terminals only, and not on other terminals or non-terminals (e.g. context) next to some non-terminal. For instance, the previously shown example of a grammar is in fact a context-free grammar.

Table 2.6: EBNF operators, and their function.

Operator			*Function*
<a>	‘|’	<b>	Choice operator, which specifies that either <a> or <b> is a valid alternative.
<a>	‘ ’	<b>	Concatenation operator (empty space), which specifies the concatenation of <b> after <a>.
<a>	‘?’		Optionality, which specifies that <a> is allowed to be included, or left out.
<a>	‘*’		Kleene-star operation, which specifies zero or more consecutive repetitions of <a>.
<a>	‘+’		Alternative repetition operator, specifying that <a> should be repeated one or more times.

2.8.2 Extended Backus-Naur Form

One of the more popular grammar normal forms for specifying context-free grammars for parsers, is the Extended Backus-Naur Form (EBNF). Grammars written in EBNF form a set of production rules, which specify how all valid strings from a given language can be generated, starting from the *root symbol S*. An EBNF production rule has the following form:

<rulename> ::= <rulebody> ;

where the rule body can contain other rules, or terminals. By defining multiple rules, the grammar can be specified in a structured and intuitive manner. Furthermore, EBNF introduces special operators, which we list in table 2.6. These operators are used to define more complex patterns of rules and non-terminals.

2.8.3 ANTLR4

ANTLR4 (ANother Tool for Language Recognition), is a tool for generating parsers for the class of ALL(*) grammars[64]. The design goals of ANTLR4 are focused on ease-of-use in favour of speed. The generated parsers however, are able to efficiently parse grammars in practice.

The ANTLR4 file format is analogous to an EBNF grammar, with additional instructions for the generated code. ANTLR4 is able to generate a parser program which is able to recognize whether a string conforms to the specified language. Additionally, ANTLR provides the parse tree which represents the rule application choices of the parser. Another useful feature is the support for labelled rules, which allows the definition of a label for terminals or non-terminals in a rule, which will be present in the parse tree. We use this feature to label the arguments of our binary expression rules, which provides us convenient access to these arguments.

In order to smooth the integration of generated ANTLR parsers in existing projects, ANTLR is able to generate listener interfaces. These interfaces allow existing code to be notified of the parsing of elements from the language, and allows the code to access and query the generated parse tree. When compared with the previous version, ANTLR3, the parse tree of ANTLR4 is better accessible. A common usage scenario which has been improved is the representation of nodes formed by the * or + operations. The ANTLR4 tree nodes representing that operation provide a list interface to such rules, which improves the accessibility of the information in these nodes.

2.9 Conclusion

In this chapter, we have discussed the tools and technologies which we use in our work. We have introduced CySeMoL and thoroughly explained the modelling features it provides, and the details of its vulnerability analysis. In the following chapter, we will use Model-Driven Engineering principles for the construction of models which will capture the information used for our new vulnerability analysis. Furthermore, we construct a model for the representation of a ProbLog program, for which we have developed a model-to-text transformation which is able to generate a valid ProbLog program from such a model. In order to reconstruct the vulnerability analysis of CySeMoL in ProbLog, we apply a model-to-model transformation from our vulnerability analysis models to our ProbLog model.

The existing CySeMoL vulnerability model contains a specification of how the vulnerabilities should be derived from a network infrastructure model. This specification is written in a probabilistic programming language (P^{2}AMF), which we automatically parse using ANTLR. The resulting parse tree is used to include the derivation in our own vulnerability analysis model.

Chapter 3

The Probabilistic Vulnerability Analysis model

3.1 Introduction

Following our first goal (see section 1.3), the improvement of the analysis speed of the vulnerability analysis of CySeMoL, we have investigated an alternative method of analysis using ProbLog. The details on how the probability of a successful attack can be inferred for a given network infrastructure is defined in the threat model used by CySeMoL. We aim to reuse the threat model of CySeMoL as much as possible, given that it has been developed as a result of years of research[36]. CySeMoL and its analysis have been documented in published articles, and the documentation of EAAT is publicly available from the KTH website[37, 36, 35, 43, 7]. However, both the analysis and the metamodel of CySeMoL depend on the availability of the EAAT framework, and are not accessible or usable outside of the EAAT ecosystem.

From our investigation of `iEaat` files, the storage format used by EAAT, we have derived a method which allows us to access the threat model of CySeMoL outside of EAAT. For a description of the `iEaat` format, and how we are able to extract information from these files, see section 4.2. Guided by these results, we have to choose between the following two approach options:

1. Transform the information from an `iEaat` file directly to a ProbLog program.

2. Use an intermediate pivot model, and transform the information from an `iEaat` file to this model. Next, define an additional transformation to ProbLog.

We have opted to choose the approach as described in item 2, the rationale for this choice is twofold: First, we aim to achieve our second and third goals (see section 1.3) which prescribe both the input to the vulnerability analysis, as well as the analysis itself to be extensible. The approach of item 2 has more potential for achieving these goals, as each step can be implemented by different means. Second, the `iEaat` file contains a deliberate separation between the definition of the vulnerability analysis, and the definition of network infrastructure under analysis. We foresee that for real networked computer systems, the network architecture may change rapidly, whereas the component definitions and the threat model will be subject to change over longer time periods. The approach in item 2 allows to retain this separation, which potentially will result in a more flexible and faster method for performing the vulnerability analysis of CySeMoL using ProbLog.

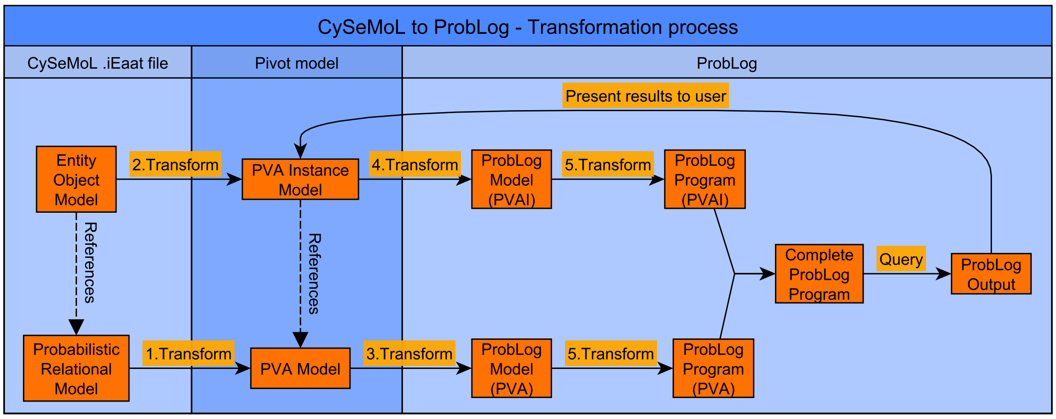


Figure 3.1: A schematic overview of the transformation process of transforming CySeMoL models stored as `iEaat` files into ProbLog programs. Details of the transformations steps are shown in table 4.1 and in chapter 4, where we discuss the design of the model transformations.

In figure 1.1, we show our design of the process for extracting the information stored in `iEaat` files into a pivot model, and how the pivot model is transformed to a ProbLog program which is used to perform the intended vulnerability analysis. Due to the separation between the specification of the vulnerability analysis and the definition of the input network infrastructure, we have decided to split the pivot model into two models. We refer to the vulnerability specification model as the *Probabilistic Vulnerability Analysis* (PVA) model, and to the network infrastructure instance model as the *Probabilistic Vulnerability Analysis Instance* (PVAI) model. Ultimately, the PVA and PVAI models are transformed into a ProbLog program. In order to reduce the complexity of our transformation scripts, we employ an additional intermediate model which represents ProbLog programs. We have developed a model-to-text transformation which generates ProbLog programs from instances of the ProbLog model.

The complete system of models and model transformations is shown in figure 3.1. This figure shows the parts of a CySeMoL model stored as an `iEaat` file on the left hand side, and the output of ProbLog on the right hand side. The rest of the figure shows all the intermediate steps of the process which we employ to obtain ProbLog programs when starting with CySeMoL models. In this figure, we also show the role of the PVA and PVAI models as a pivot model; changing the vulnerability analysis amounts to the replacement of the 'ProbLog' block of models and model transformations.

In this chapter, we discuss the details of the design of the PVA, PVAI and ProbLog models, and the design choices made during the development of these models. We cover the design of the transformation scripts in chapter 4, and we discuss the implementation of these scripts in chapter 5.

3.2 Probabilistic Vulnerability Analysis model design

3.2.1 Overview

CySeMoL can not easily be used outside of the EAAT environment, which obstructs the possibility to extend its analysis or replace it with alternative methods. In order to retain these abilities, we choose to define our own model using a model-driven approach. Concretely, model-driven entails that we define the PVA, PVAI and ProbLog models using metamodelling. The aforementioned models are defined as EMF models in the *Ecore* format (see sections 2.4 and 2.5). By using EMF and the industry-standard `Ecore` format, we greatly enhance the extensibility and usability properties of our models.

The PVA model is designed to capture the semantics of EAAT class models, tailored to the class model defined by CySeMoL. In practice this means that, using the PVA model, we are able to represent the current definition of CySeMoL (version 2.3), including all assets, defences, attack steps, and derivations. Using the PVA model, we are able to extract and store the definition of the threat model of CySeMoL from `iEaat` files.

During the discussion of the PVA model, it can be helpful to review the class diagrams listed in appendix A.2. These diagrams provide a schematic overview of the structure of the PVA model. Due to its structural size, the model diagrams have been split into functional units in order to improve their readability.

3.2.2 Structure

In our PVA and PVAI models, we want to support the EAAT workflow which is used for the modelling of network infrastructures and the analysis of their vulnerabilities. In P^{2}CySeMoL, related assets, attack steps and defences have been grouped into 'templates' (see also section 2.3.3). During the modelling of the network architecture, structures of these templates are composed together in order to construct a valid model of assets, and their associated attack steps and defences. With this use in mind, we have divided the PVA model into three parts:

1. The concrete part, which is used to model network components.

2. The template part, which groups of concrete parts in the form of templates.

3. The probabilistic expression part, which is used to model the P^{2}AMF code.

The three parts are joined together by the `PVAContainer` class, which serves as the root of the model. The PVAContainer class holds a reference to all concrete and template classes.

All top level classes (those classes with `PVAContainer` as a parent) implement the `PVAType` interface. This interface ensures all implementing instances are uniquely identifiable through the use of a 64-bit UUID. In addition, it is possible to assign a human-readable name. The design choice was made to allow names to occur more than once, while still maintain support for correct XMI serialization (see also section 2.5). A common scenario in which the use of these UUIDs is required, is with templates and asset naming: as CySeMoL defines many (10+) templates with the same name as an asset, instances for these classes are not globally identifiable by their name only.

3.2.3 Individual model components

CySeMoL is defined using the rich feature set of EAAT. However, for our goals we are only required to model the vulnerability analysis provided by CySeMoL. For this reason, we refrain from defining a model for the representation of arbitrary EAAT models, but focus on CySeMoL features instead. Consequently, explicitly model *Assets*, *Defences*, *Attack Steps*, and the *Attacker*, instead of modelling EAAT classes.

The *Assets* model the core network components and systems. In the context of CySeMoL, these components can range from individual systems to users, networks, data flows, and office spaces. The classes representing such assets are modelled as the `AssetType` class in the PVA model.

These assets can be attacked by performing a sequence of *Attack Steps* (modelled as the `AttackStepType` class). An `AttackStepType` always has one `AssetType` as its target. Some attack steps are only available when some prerequisite attack step has been completed first (the case where multiple attack steps need to be completed is modelled through the P^{2}AMF code). This feature is modelled using the 'enables' edge. In CySeMoL an attack step being enabled means that its code for determining its success can be evaluated. We store this code in the probabilistic property which is contained in the 'succeeded' edge. For details on how the probabilistic code is modelled in our PVA model, we refer to section 3.2.5.

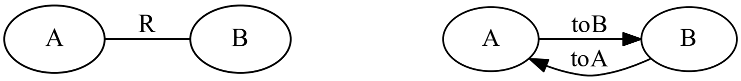


Figure 3.2: The example relation *R* between classes A and B, and their respective views.

For the purpose of modelling of countermeasures to attacks, CySeMoL has the notion of *defences*. We model defences using the `DefenceType` class. In CySeMoL, whether defences are functioning is a binary state, ergo defences are either functioning or they are not. Furthermore, defences can become less or more effective, depending on external factors such as completed attack steps. These properties are all modelled in the same fashion as attack steps, by the probabilistic property contained in the 'functioning' edge.

EAAT implements class inheritance, which has the same semantics as Object Oriented inheritance. CySeMoL leverages the inheritance support for the implementation of default behaviour of operations. In particular, defences are specified as having a 50% probability of functioning by default. And attack steps succeed by default if they are enabled. In the PVA model, this behaviour is modelled by defining the implementation of probabilistic properties as optional. The intended semantics are such that when the implementation is absent, the default behaviour is generated. When an implementation is not absent, the behaviour specified by the probabilistic property is used.

3.2.4 Connections

EAAT supports the definition of bi-directional relations between arbitrary classes. In the PVA model, we chose to implement this feature using uni-directional edges. The class used for edges between attacks, defences, and attack steps is the `ConcreteConnection` class. The rationale for this design choice is that uni-directional edges are more flexible, given that bi-directional edges can be trivially implemented by using two uni-directional edges instead.

The bi-directional relations from EAAT support different labels for each relation, depending on the class at each end of the relation. Consider the example of figure 3.2, where we have defined a relation *R* between two classes A and B can have be navigated from A to B by the label 'toB', and from B to A by the label 'toA'. To be able to refer to the local navigation labels, connections in the PVA model always have a 'label' field, which reflects this feature from EAAT. The direction of each label is stored in the model by assigning the connections with the correct labels to their respective owners.

Another feature from EAAT are the so-called 'derived' connections, which consist of a sequence of other existing relations. These derived connections are used to simplify the navigation of long or complicated paths of relations in a class model. Derived edges are also bi-directional, however, the path of relations is specified from the perspective of the owner of the relation. We implement these derived edges in the `VirtualConnection` class, which stores a list of the relations which have to be traversed. Given our design choice for uni-directional edges, we encounter a slight problem here, as we have to traverse the path in reverse for one end of the relation. We solved this problem by storing whether the path of a single virtual connection has to be traversed in reverse using a boolean variable.

3.2.5 Probabilistic expressions

In the PVA model, the 'functioning' and 'succeeded' properties of the `DefenceType` and `AttackStepType` classes represent probabilistic properties. In CySeMoL these are the `AttackStep.likelihood` and `Defense.functioning` properties, which are specified using P^{2}AMF. In the implementations of the CySeMoL properties, we noticed the following recurring patterns:

- Probability distributions are defined before any other code through '`let ... in`' declarations.
- Probability distributions are always referred to by their name, and are never defined inline.
- A check whether some defence is functioning always has the same syntactical form, all such checks are always specified in the exact same way.
- A check whether some attack step has succeeded always has the same syntactical form.
- The code sometimes contains null checks for some reference, the target of these references are always assets.

We verified whether these observations hold for all P^{2}AMF code blocks in the P^{2}CySeMoL model by writing a parser grammar conforming to these observations. In sections 4.3.3 and 5.3.5, we tread into detail on the development and specification of these grammars. The generated parser from the grammar was able to correctly parse all P^{2}AMF code blocks, which confirms that these observations apply to all current P^{2}AMF code of CySeMoL.

We model the implementation of the probabilistic properties as single expressions, which are allowed to use an arbitrary number of named probability distributions. The expressions may contain references to existing concrete types, where the scope of the reference is the owner of the probabilistic property. The result of the expressions is a number in the range $[0,1] \subset \mathbb{R}$, which maps the probabilistic property to the probability that the property will be evaluated to 'true'.

A probabilistic code block may define an arbitrary number of independent probability distributions which are (implicitly) parametrized by the amount of time an attacker has. These distributions are identified by a unique name within the rest of the code. Our current model supports the probability distributions used in P^{2}CySeMoL (see section 2.3.7). This amounts to the following types of probability function:

a) The Bernoulli distribution.

b) The Exponential cumulative distribution function.

c) The Exponential probability density function.

d) The Normal probability density function.

e) The Log-normal cumulative distribution function.

f) The Gamma cumulative distribution function.

The metamodel of the probabilistic code partially reflects the abstract syntax tree of the P^{2}AMF code. We define the following operands:

g) Constant probabilities, this includes 0.0 for 'false' and 1.0 for 'true'.

h) One of the declared probability distributions.

i) A reference to some defence, which tests whether this defence is functioning.

j) A reference to some attack step, which tests whether this attack step has been completed.

k) A reference to some asset, which tests whether this asset can be obtained by following the provided reference.

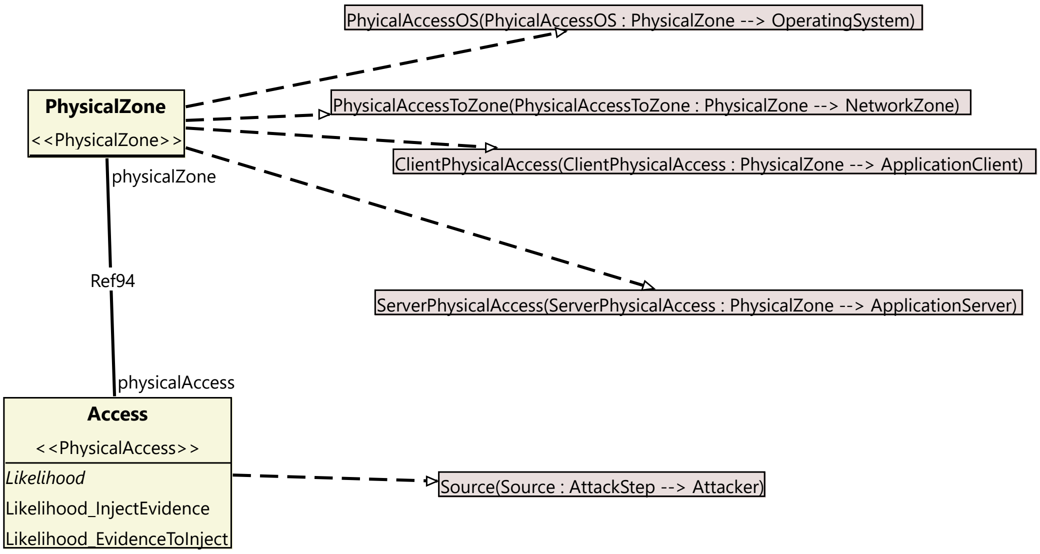

PhyicalAccessOS(PhyicalAccessOS : PhysicalZone --> OperatingSystem)
PhysicalZone
<<PhysicalZone>>
PhysicalAccessToZone(PhysicalAccessToZone : PhysicalZone --> NetworkZone)
ClientPhysicalAccess(ClientPhysicalAccess : PhysicalZone --> ApplicationClient)
physicalZone
Ref94
ServerPhysicalAccess(ServerPhysicalAccess : PhysicalZone --> ApplicationServer)
physicalAccess
Access
<<PhysicalAccess>>
Likelihood
Likelihood_InjectEvidence
Likelihood_EvidenceToInject
Source(Source : AttackStep --> Attacker)

Figure 3.3: A demonstration of the definition of the `PhysicalZone` template of CySeMoL as displayed in the EAAT class modeller. Note that the small gray boxes represent potential external connections. For example, the `Access` instance can be connected through a 'Source' edge to an instance of the class `Attacker`.

We can combine the leaf nodes using one of the combining elements. These are the following:

l) Probabilistic `And`, which returns the probability that both leaves evaluate to 'true'.

m) Probabilistic `Or`, which returns the probability that at least one of the leaves evaluates to 'true'.

n) Probabilistic `Not`, which returns the probability that its leaf evaluates to 'false'.

o) Probabilistic `If-Then-Else`, which returns the probability that both its condition and the `true` leaf will evaluate to 'true' or that its condition will evaluate to 'false' and its `false` leaf evaluates to 'true'.

The aforementioned components are sufficient to express the P^{2}AMF code which occurs in P^{2}CySeMoL. This was verified by transforming the P^{2}AMF code to the PVA model. This specific part of the transformation is described in section 4.3.3.

3.2.6 Templates

On top of the concrete vulnerability analysis classes (those classes which subclass `Asset`, `AttackStep` or `Defense`), CySeMoL utilizes the 'template' feature of EAAT. This feature introduces the definition of collections of interconnected class instances. In the object modeller, instances of these templates can be defined and connected to other templates or class instances. CySeMoL uses these templates to conveniently define complex structures of attack steps, assets and their defences. This facilitates the construction of models from reusable components. In addition, the templates are used to filter the displayed attack steps for their corresponding asset. For example, intermediate attack steps are often hidden to improve the readability of the analysis results. In figure 3.3, we show the contents of the `PhysicalZone` template definition as they are displayed in the EAAT class modeller. When used in the object modeller, the concrete template is displayed as shown in figure 3.4.

We have implemented our interpretation of the templating feature from EAAT in the PVA model as well. Our interpretation of templates defines a template as a structure which specifies an arbitrary number of concrete types which may or may not be connected to other concrete types. The intended purpose of the templates in the PVA model is to generate a collection of class instances along with their connections during the analysis for each template instance.

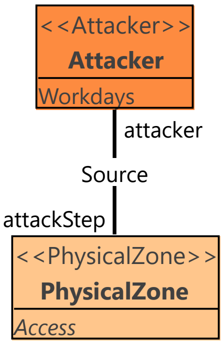


Figure 3.4: The `Attacker` and `PhysicalZone` template as displayed when used in the EAAT object modeller. Note that the 'Source' edge represents a relation between class instances defined in both templates.

The PVA model distinguishes intra-template connections from extra-template connections. The extra-template connections mimic the 'port' feature from EAAT, which allows templates to support connections to classes outside templates. The classes which are responsible for the intra-template connections are the `TypeInstantiation` and `InstantiationConnection` classes. When a template instance is defined in the PVAI model, it denotes the generation of instances represented by these two classes. To facilitate this, these instantiation classes keep a reference to the concrete type the generated instances will have. In other words, the 'concretization' edges specify the concrete type of the generated instances during the analysis. The extra-template connections are specified as connections to concrete types, in the same fashion as the ports from EAAT (we will demonstrate how these external connections are populated when we introduce the PVAI model in section 3.3).

EAAT allows to specify arbitrary properties from entities inside templates as 'template attributes'. This causes these attributes to show up in the object modeller (for example, see the `likelihood` property of the `PhysicalAccess` attack step in figures 3.3 and 3.4). This feature is used by CySeMoL to display the success probability of attack steps and the likelihood that defences are functioning. The only attributes referenced by CySeMoL happen to correspond to our probabilistic properties, so it suffices to include a reference to these properties in the PVA model.

3.3 Probabilistic Vulnerability Analysis Instance model design

3.3.1 Introduction

In the previous section, we have laid out the general design of the Probabilistic Vulnerability Analysis model. In the PVA model, we have included the ability to specify templates which are used to model collections of assets, attack steps and defences. The PVA model is designed to the specification of the components which constitute network infrastructures.

In order to facilitate the definition of concrete network infrastructures, we have developed the Probabilistic Vulnerability Analysis (PVAI) model. The PVAI model serves as an intermediate model for modelling the information specified using the *Object Modeller* from EAAT (see section 2.3.3). Such a concrete infrastructure is defined as a set of instances of templates, which may be interconnected.

In this section, we will discuss the design of the PVAI model. Similar to the PVA model, we have included the class diagrams of the PVAI model in appendix A.3. We have also developed a method which is able to create PVAI models from the information stored in `iEaat` files. The details of this method will be discussed in section 4.4.

To maintain the consistency of PVAI models, we require that a set of model invariants is enforced. These invariants and their definition are listed in table 3.1. When we require one of the invariants, we will refer to the names as listed in this table.

Table 3.1: The PVAI model invariants

n	*Entity label*	*Invariant*
INV1	EntryPoint : E	E.target ∈ E.targetTemplateInstance.definition .instantiations
INV2	Evidence : Ev	Ev.target ∈ Ev.propertyOwner.properties
INV3	ConnectionTemplate : C	C.source ∈ C.eContainer.instantiations
INV4	ConnectionTemplate : C	C.target ∈ C.targetOwner.instantiations
INV5	ConnectionTemplate : C	C.definition ∈ C.source.concretization .connectedTo
INV6	ConnectionTemplate : C	C.definition.target == C.target.concretization

Similar to the PVA model design, we have defined the PVAIContainer class, which serves as the root container class for PVAI models. The root container holds references to all top-level classes through the PVAIType interface. This interface is also used for objects which must be uniquely identifiable. In this sense, the PVAIType interface is similar to the PVAType interface from the PVA model. Furthermore, we use a unique 64-bit unsigned integer is used for identification, and an optional name to enhance the readability for humans.

In addition to the classes implementing the PVAIType, the root container also maintains a reference to a single Attacker object. We observe that every meaningful analysis in CySeMoL requires an attacker. In our model, we generalize CySeMoL's analysis to a single attacker, we will discuss our rationale for this decision in section 3.3.5.

3.3.2 Template instances

Central to the usability of EAAT is the definition of templates (as discussed earlier in section 3.2.6). In CySeMoL it is possible to create models using a mixture of templates and concrete classes. However the use of concrete classes is prone to human error, as it is possible to create incomplete or invalid models. Therefore, in order to prevent modelling mistakes, we enforce the use of templates in the PVAI model by not supporting concrete classes in the instance models. An additional advantage is that templates are able to encapsulate complex interactions between components, and filter only the relevant information for users.

As mentioned before, a PVAI model consists of a set of template instances, along with an attacker. The TemplateInstance class reflects the instantiation of the template defined in a TemplateDefinition object. The template instance holds a reference to its definition object through the 'definition' edge. The instantiation of a template comprises the creation of instances for all TypeInstantiation objects of the template definition. By the use of an additional TemplateInstance class, we allow the instantiation of multiple templates of the same type. Usage within CySeMoL of this feature is demonstrated in figure 2.6. EAAT allows evidence to be defined on a per-template basis. To implement this, the TemplateInstance class also keeps track of its specified evidence through the 'evidence' relation.

3.3.3 Template connections

In CySeMoL it is possible to connect template instances. This is performed by drawing lines between templates, and selecting the desired relation if multiple types of relations are allowed. These connections actually reflect connections between instantiations instead of the templates themselves. This is also displayed in the Class Modeller of EAAT, as shown in figure 3.3.

In the PVAI model, connections between templates are associated with the source template. The model class reflecting this is the `ConnectionTemplate` class. To maintain coherence with the CySeMoL semantics, we also store the definition, source and target of the template connection. Note that we could also model the template connection as an association class on the relation between two template instances. However, the framework in which we define the PVA and PVAI models (EMF, see section 2.5) does not support association classes, which means that the current design with 1 incoming and 4 outgoing edges is a workaround, resulting in a substandard class diagram. In order to maintain model consistency we define invariants INV4, INV5, INV6, and INV7, to ensure that all defined template connections remain valid.

3.3.4 Evidence

Another important feature of CySeMoL is reasoning under evidence. We interpret this feature as casting the outcome of a probabilistic property to a value of either zero or one. Effectively, this means that the evidence is assumed, and that all other probabilities are conditioned with the evidence.

We restrict the use of evidence to the following two cases:

1. Success or failure of an attack step.
2. Whether a defence is functioning or not.

In our metamodel, this is reflected by the `Evidence` class, which has a `ProbabilisticProperty` instance as its target. Due to our design, the only uses of this class are for the two cases for which we support evidence. The probabilistic property is owned by the `TypeInstantiation` contained in the 'propertyOwner' field of the `Evidence` class. The ownership is enforced by INV3. The concrete outcome of the evidence is contained in the `Evidence` class itself, as a boolean attribute.

3.3.5 Entry points for the Attacker

CySeMoL allows to define multiple attackers, although it does not support multiple entry points of the same type. This effectively restricts models to the definition of entry points, instead of a number of hostile individuals. Furthermore, the analysis of P^2CySeMoL does not distinguish the individual attackers during the analysis. The only advantage of allowing multiple (independent) attackers, is the ability to take into account different time investments by each individual attacker.

We set our goal to be more independent on the 'workdays' parameter, which limits all attacks to the same exact time-frame. To this end, we decided to model a single attacker only. When an input CySeMoL models contains multiple attackers, we model their entry points as originating from a single attacker. In our model, the attacker is modelled through the `Attacker` class, which carries an identifying name, and a reference to entry points. Performing our analysis is not meaningful when there are no entry points (the probability of the attacker succeeding at anything would be zero), likewise we require the definition of at least one entry point.

We model an entry point as some initial attack step which has already succeeded. We model this in the `EntryPoint` class, where we store the attack step type, and some type instantiation within a template. As this is the PVAI model, we refer to a template instance. To keep the model correct, it must be invariant that the `EntryPoint.target` reference must be defined in `EntryPoint.targetTemplateInstance` template, ergo, for an entry point the invariant is as shown in INV2.

3.4 The ProbLog model

3.4.1 Introduction

In this thesis, we define a method for performing the vulnerability analysis of CySeMoL using ProbLog. In order to achieve this, we have opted to use a model-based approach, where the PVA and PVAI models function as a pivot model. Next up is the ProbLog model, which we will use as the target model for representing the probabilistic vulnerability analysis. The model needs to be suitable for model-to-text transformations, as we plan to perform such transformations after the transformation from either a PVA or PVAI model has been completed. For the class diagram of the ProbLog model, we refer to appendix A.4. The transformation to ProbLog, and the implementation of the vulnerability analysis will be discussed in sections 4.5 and 5.4.

We designed the ProbLog model to contain only low level information, without any knowledge on the vulnerability analysis. Consequently, the ProbLog model is also applicable to different usage scenarios. The ProbLog model cannot be directly used as an input to the ProbLog inference tool. Therefore, we use a model-to-text transformation which generates a valid ProbLog program from a ProbLog model. The details of this transformation are discussed in section 4.5.4.

3.4.2 Program structure

We model a ProbLog program as a sequence of statements, represented by the `Statement` interface. There are two main types of statements, *rules* and *ProbLog statements*. Rules model arbitrary ProbLog rules, and a ProbLog statement is either a query or an evidence specification.

Queries are specified using the `Query` class, which behaves similar to a regular term instance with a single argument. The `Evidence` class models the special 'evidence' predicate of ProbLog, which is used to assign either 'true' or 'false' to existing probabilistic rules. Both queries and evidence are specified using a single statement, which is stored in the 'subject' property of these classes. Furthermore, we include some auxiliary statements which do not directly influence the logic rules. The `Comment` and `ImportLibrary` classes represent such statements. Respectively, they model single-line comments and the importing of libraries.

We model rules as having a left-hand side (LHS) of possibly annotated disjunctions, and having an optional right-hand side (RHS) of conjunctions. We have explicitly chosen not to support disjunctions in the RHS, as the same semantics can also be achieved by adding multiple rules with an equal LHS instead.

The combination of a term and its arguments plays a special role in ProbLog programs. This combination is used as a target in queries and may appear in both the LHS and RHS of a rule. Considering this use and scope of terms, we specify that terms must be declared at the program level using the `Term` class. References to terms are constructed by means of the `TermInstance` class, which holds a reference pointing back to the `Term` of which it is an instance.

3.4.3 Annotated disjunctions

Following the Prolog specification, we only allow atoms or term instances on the LHS. In order to annotate only these types, we introduced the `Annotatable` interface. Annotations are specified using the `AnnotatedReferable` class, which wraps an `Annotatable` instance.

The corresponding probability values are specified using the children of the `ProbabilityMeasure` (abstract) class. We explicitly model *fractions* and (double-precision) *literals*. The `AnnotatedReferable` holds a reference to such a definition to keep track of the probability assigned to it.

3.4.4 Referable entities

Entities which are allowed to occur on the right-hand side of a rule, as arguments of lists, tuples and term instances, are grouped under the `Referable` interface. The following types can be referenced, and implement the `Referable` interface:

- The `TermInstance` class, which introduces support for nested terms.
- Variables through the `Variable` class.
- Atoms through the `Atom` class.
- Special collection types for which ProbLog has built-in support, specifically *tuples* and *lists*. These types are implemented using the `PLList` and `PLTuple` classes.

By creating compositions of referable entities, which may also refer to other referable entities, we support the definition of complex combinations of deeply nested language elements. A common use-case for this feature is the nesting of predicates, in which some predicate is passed as an argument of another predicate.

Chapter 4

Transforming P^2CySeMoL models

4.1 Introduction

In the previous chapter, we have introduced the PVA, PVAI and ProbLog models. In order to reproduce the vulnerability analysis provided by CySeMoL with an alternative methods, we use the PVA and PVAI models as pivot models which store the information contained in `iEaat` files. Our alternative analysis method is based on probabilistic logic (ProbLog) instead of CySeMoL. We transform the information contained in a pair of PVA and PVAI models to a ProbLog program, which we use to infer the likelihood that an attacker succeeds in compromising components of the network infrastructure defined in the PVAI model.

In this chapter, we will discuss the method which we designed to transform the problem of Monte Carlo estimation of CySeMoL models using P^2AMF, to the problem of inferring probabilities in probabilistic logic programs using ProbLog. A schematic overview of the entire process is shown in figure 3.1. In the previous chapter, we have discussed the design of the PVA, PVAI and ProbLog models. Recall the outline of the transformation process as shown in figure 3.1, the first step of which is the extraction of the CySeMoL information stored in `iEaat` files into instances of the PVA and PVAI models. In this chapter, we will outline the process which we developed to perform this transformation of a CySeMoL model into PVA and PVAI models.

In section 4.2, we begin with the introduction of the file format used to store CySeMoL models, and then define the transformation to the PVA and PVAI models. From those models, we derive a ProbLog program which reflects the analysis of CySeMoL which we will discuss in sections 4.5.2 and 4.5.3. Finally, we will tend to the concern of the orchestration of the transformation process between all these models.

In total, we have constructed five different model transformations. Furthermore, these model transformations have been implemented with different underlying technologies. In table 4.1 we list the transformations, their type, the technologies used to implement the transformations, and an estimate of their complexity indicated by the total lines of code used for their specifications.

Table 4.1: An overview of all transformations constructed for this research. In this table, we list the source and target models, the technologies used to implement the model transformations, and the complexity of the transformation scripts.

#	Source model	Target model	Type	Technologies	LoC
1	CySeMoL PRM	PVA	Model-to-model	Kotlin + ANTLR4	875 + 188
2	CySeMoL EOM	PVAI	Model-to-model	Kotlin	351
3	PVA	ProbLog	Model-to-model	Kotlin + ETL	34 + 1126
4	PVAI	ProbLog	Model-to-model	Kotlin + ETL	36 + 397
5	ProbLog	ProbLog Program	Model-to-text	Kotlin + EGL	40 + 138

4.2 iEaat file format

The starting point of our transformation process are CySeMoL object and class models. These models are generated by the EAAT tool, and can be exported to the local filesystem.

CySeMoL projects are saved using the `iEaat` format. By analysing the exported files, we conclude that `iEaat` files conform to the following structure: an `iEaat` file is a compressed archive (ZIP v2.0), which contains all required files for the editing and analysis of CySeMoL models. The files contained in the archive can be the following (based on the file extension):

- `EOM`, a file in the XMI format, containing the specification of the defined network architecture.
- `PRM`, a file which contains a serialized Java object. After examination using a Java debugger, this object seems to contain the class model, including the specification in P^{2}AMF on how the probabilistic analysis is performed for each attack step and defence.
- `amxCanvas` files specify the view definitions for the Class Modeller in XML format.
- `cmxCanvas` files specify the view definitions Object Modeller in XML format.

With the goal of providing easier access to the internals of `iEaat` files, we wrote a small library using Kotlin. This library provides access to a stream to the `EOM` file, and a method to deserialize the Java object stored in the `PRM` file. For this last step, we require a class definition of the `ClassModel` object. Fortunately, EAAT includes `jar` libraries with these definitions. After performing these steps, we were able to obtain a more usable interface to the class and object model stored in `iEaat` files.

In order to implement our model-based approach to the construction of an alternative vulnerability analysis for CySeMoL models, we extract the information stored within `iEaat` files. Using this information, we construct PVA and PVAI models which are able to represent the threat model of CySeMoL. In the next section, we will discuss how we use this interface to the `iEaat` files to achieve this.

4.3 Deriving PVA models from PRM files

4.3.1 Design considerations

The `iEaat` files from the previous section contain both an object model and the CySeMoL class model. This separates the definition of network components from the definition of instances of these components. In our own models, we also maintain this separation wherever possible. In this section, we will discuss the transformation to the PVA model, and we will explain the transformation to the PVAI model in section 4.4.

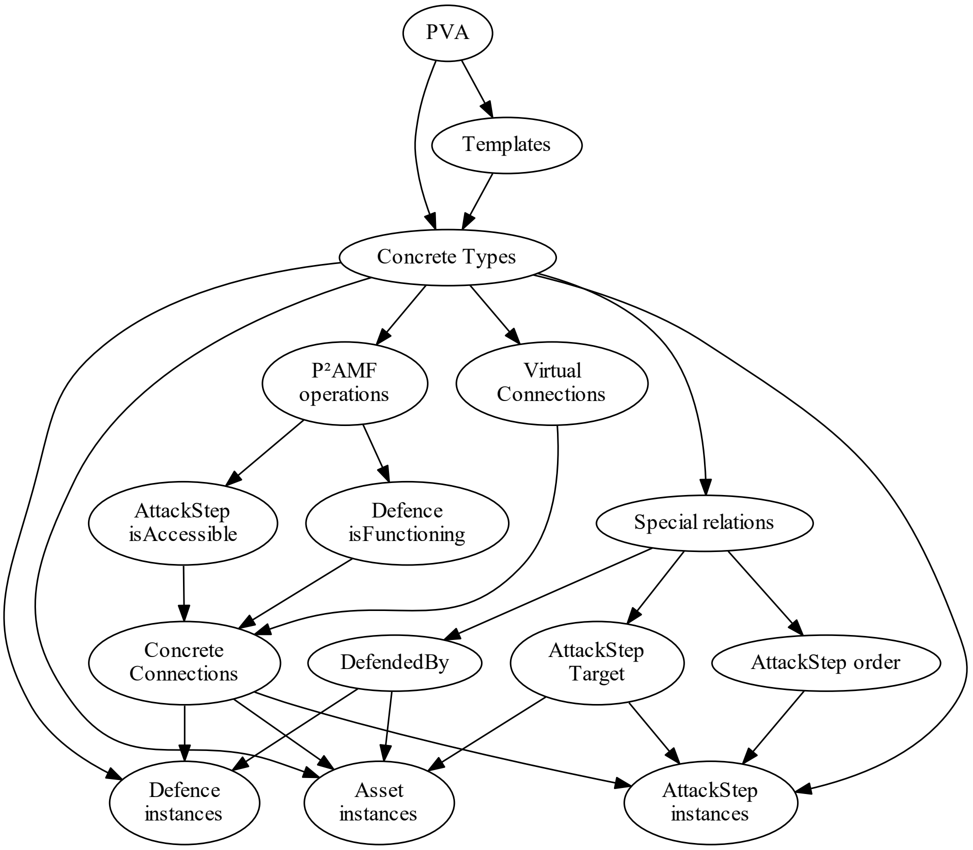


Figure 4.1: A graph representing the construction dependencies of the PVA model. The arrows indicate a dependency on the existence of another object.

4.3.2 Derivation order

In order to construct our PVA model instances, we will need to extract the relevant data (i.e. the class model) from the PRM files contained in the iEaat files. For the implementation details on how we achieve this, see section 5.3. The result of our efforts with regard to the parsing of iEaat files, is that we end up with a Java object representation of the class model of CySeMoL. Using the code-generation feature from EMF, we first generate (Java) classes conforming to our PVA model. During the transformation from a PRM file, we use these classes to construct a PVA model in-memory. Afterwards, serialize the resulting PVA model using XMI (see section 2.5.4), and pass the serialized model to the next transformation stage.

We have implemented a program in Kotlin(see also section 5.1) which constructs a PVA model from the CySeMoL class model by using the aforementioned generated EMF classes. Due to the design of the PVA model, there are some difficulties in the transitive creation dependencies, that is, the creation order of valid objects is complicated by dependencies in their relations. For this reason, we opt to create partial objects during runtime, which are completed when applicable. We determine the creation order in such a way that, when a reference is required, the corresponding object (which might be incomplete) *can always be identified*. The construction dependencies are as shown in figure 4.1.

From these construction dependencies we derived the following order suitable for the construction of PVA models:

1. The model container
2. Concrete Types
    - (i) Assets
    - (ii) Defences
    - (iii) Attack steps
    - (iv) Concrete connections
3. Templates
    - (i) Template definitions
    - (ii) Type instantiations
    - (iii) Internal connections
    - (iv) External connections
4. Special relations
    - (i) `AttackSteps.target` relation
    - (ii) `Asset.defendedBy` relation
    - (iii) Attack step dependencies
5. P^{2}AMF operations
    - (i) `AttackStep.isAccessible`
    - (ii) `Defense.isFunctioning`
6. Virtual connections

The data for the components is derived from the `PRM`, by traversing the class model it contains. In section 5.3 we will go into detail on this process, and will show the specific parts of the `PRM` where the data for our PVA model elements is obtained. The end result of applying the construction ordering is a complete and valid PVA model.

4.3.3 Parsing P^{2}AMF code

In this section, we will discuss how we derive the P^{2}AMF code for the P^{2}AMF operations stored in the PRM. These operations are stored in the PRM as strings, which are parsed during our transformation to the PVA model. By analysing the structure of the P^{2}AMF code of CySeMoL, we noticed that the most code blocks follow a common pattern. For this reason, we have developed a parser for the P^{2}AMF code in CySeMoL using the ANTLR4 parser generator engine (for more details on parser generators and ANTLR, see section 2.8).

The code blocks which our parser targets are:

- The `AttackStep.isAccessible` operations.
- The `AttackStep.getPaths` operations.
- The `Defense.isFunctioning` operations.
- The code describing how derived edges should be resolved.

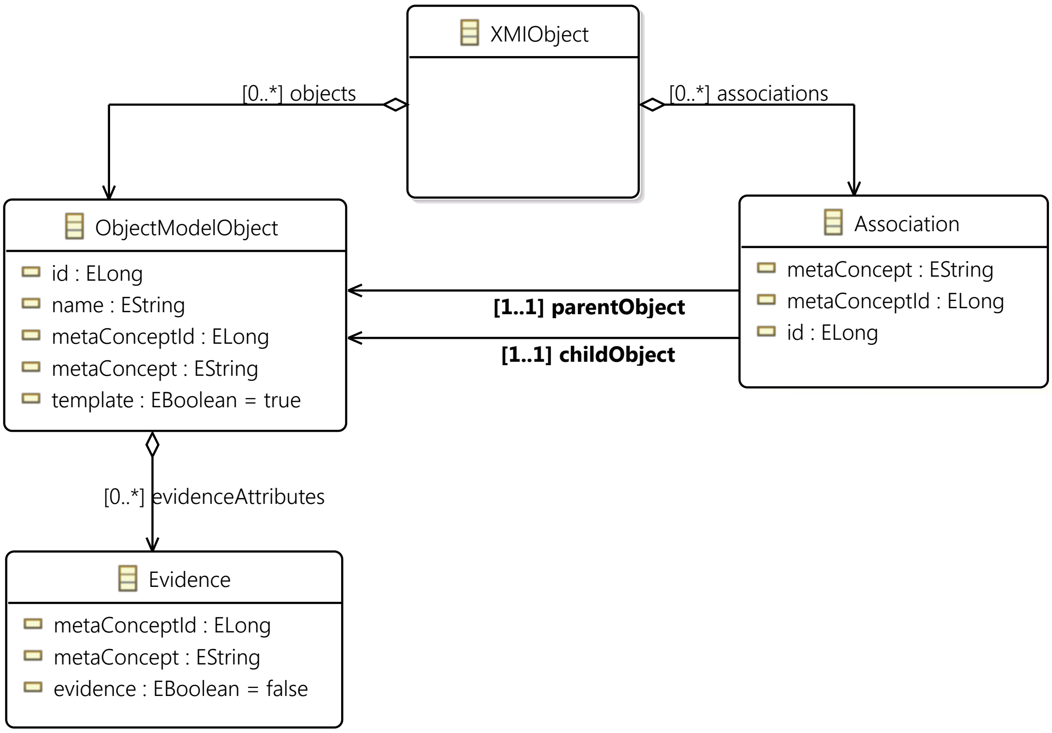


Figure 4.2: The derived structure of the `EOM` model. Displayed as an Ecore class diagram.

These code blocks are all written within the scope of the containing instance. For example, the code in `PhysicalAccess.isAccessible` is written from the scope of an instance of the `PhysicalAccess` class. For derived edges, the scope of the 'owner' of the relation is used. Given that all relations used in CySeMoL are bi-directional, this scope seems to be irrelevant. However, in the case of derived relations, the scope of the program remains unchanged when the relations are referenced by the 'child'. In this case, when resolving derived edges, the owner of a bidirectional edge is likely to be relevant. In practice, the navigation path specified by the P^{2}AMF code of these relations must be resolved in both the normal and reverse directions.

For our approach, to the parsing of the aforementioned P^{2}AMF scripts, we have developed a set of three parsers. We have constructed one large parser which is able to parse the `isAccessible` and `isFunctioning` methods, and two smaller specialized parsers which each are able to parse one of the other two P^{2}AMF code sources.

4.4 Deriving PVAI models from EOM files

The first step in deriving PVAI models from EOM files the deserialization of the information contained in those files. Although we do not have the `Ecore` model of the EOM files, the structure is simple enough to hand craft a SAX-parser[61] (a Java API for parsing XML files) for the deserialization task in Kotlin. From our analysis of the EOM files, we conclude that it conforms to the XMI format, and contains the information as if it had been defined according to the class diagram shown in figure 4.2.

Using this structure, we follow the parsing order of the SAX-parser to generate model objects for the PVAI model. We handle the XML nodes in the following way: The `objects` tag in an EOM file triggers the generation of a `TemplateInstance`. Nested within the `objects` tag can be an optional `evidence` tag, which adds evidence to the template generated for its enclosing tag. The `associations` tags are mapped to bi-directional associations between two `TemplateInstance` objects created before. By generating classes as their corresponding tags are encountered, a single pass over an EOM file results in the construction of a valid PVAI model. Similar to the PVA model, the PVAI model is serialized using XMI and passed to the next transformation stage.

4.5 Transformations to ProbLog models

4.5.1 Introduction

From the PVA and PVAI models, we systematically construct a ProbLog program. Due to the deliberate separation of concerns between PVA and PVAI models, we have implemented this transformation in two separate steps (see also the last box of figure 3.1). This results in two separate ProbLog programs, which can be merged to form a complete program. The reason for this separation is that we can quickly generate a new program when the object model changes, without generating code for the class model.

4.5.2 Transformation of PVA models to ProbLog models

We implement the transformation of PVA models to ProbLog models as a model-to-model transformation. Our transformation of a PVA model to a ProbLog model attempts to address the following two concerns:

1. The data stored in PVA models.

2. The vulnerability analysis has to conform to the analysis of CySeMoL.

For item 1, we generate non-probabilistic facts denoting the existence of the objects in the PVA model. This results in a long list of facts (in our tests using CySeMoL we generate 1636 facts in total), which we will use in the implementation of item 2, to construct the analysis.

Item 2 is implemented using a different inference approach than CySeMoL. This is due to the fact that ProbLog does not require sampling to infer the marginal probabilities of a model. Instead of sampling, we use ProbLog to infer the exact probabilities by the algorithm as described in section 2.7.6. Given that the scope of the P^{2}AMF derivations are specified relative to the attack step and defence instances, we generate ProbLog code with unique statements for each derivation. This way, we ensure that derivations of attack steps are independent from other derivations. Furthermore, we define helper functions for the probability distribution function evaluation, which return a single probability. For testing purposes, we conform to the 'workdays' parametrization as used in CySeMoL. However, by abstracting from the implementation through these helper functions, we are free to change this parametrization if we so desire.

The probability of success for an attack step instance is determined as follows: first, we let ProbLog determine which attack steps are reachable, according to the graph induced by the `AttackStepType.target` edges. For these attack step instances, we calculate the success probability by evaluating the converted P^{2}AMF code for each attack step instance. ProbLog allows us to specify that attack steps success probabilities may depend on the success probabilities of other attack steps, and manages the corresponding calculation using conditional probabilities. Due to the fact that we use parametrized probability distributions, we are able to infer the exact success probabilities of attack steps using this approach with ProbLog.

4.5.3 Transformation of PVAI models to ProbLog models

The transformation from PVAI models to ProbLog models is independent from the transformation of PVA models to ProbLog. This due to the fact that the PVAI model references its corresponding PVA model, which is accessed during the transformation process to ProbLog. Consequently, this transformation relies on the same information used by the PVA transformation.

We designed the resulting ProbLog program for the PVA program in such a way, that it is able to derive the entire analysis from the specification of template instances, their connections, and an entry point for the attacker. Therefore, most of the transformation effort is performed in the PVA transformation. This allows our PVAI transformation to remain simple and fast. Due to our implementation of the PVA transformation, the PVAI transformation comprises the generation of facts about the data in the PVAI model. We are able to trivially generate facts for:

- Template instances
- Entry Points
- Evidence

However, due to the construction of our PVAI model, the generation of connections between templates is more involved. In order to generate connections between templates, the transformation script derives the concrete relation the template connections resolve to, and creates facts which directly instantiate these *concrete* connections.

4.5.4 Transformation of ProbLog models to ProbLog programs

ProbLog requires a program written in text as its input, therefore, we apply a model-to-text transformation from our models to ProbLog code. Due to the low-level design of the ProbLog model, the model already bears structural similarities to ProbLog programs. Recall that the ProbLog model was designed with the intention to serve as an input for model-to-text transformations, which is reflected in the property that all components have local references to all other model elements required for the generation of their text representation.

We have implemented a single transformation script using EGL, which is able to transform arbitrary ProbLog models into a ProbLog program. This script is executed for both the PVA and PVAI model. The output from both transformations is merged into a single file. Finally, some auxiliary ProbLog code is appended to the resulting total program, resulting in a valid ProbLog program. Running this program will cause ProbLog to determine the success probability for all attack steps and the probability that all defences are functioning.

Chapter 5

Implementation

5.1 Introduction

In the chapter 3, we have introduced the PVA, PVAI and ProbLog models. We intend to use the PVA and PVAI models as a pivot model between CySeMoL models and our ProbLog model, as shown in figure 3.1. The use of a pivot model requires the definition of a set of model-to-model transformations to and from that model. The overall design of these transformations have been laid out in chapter 4. In this chapter, we will focus on the implementation of the transformation steps for our approach to an alternative vulnerability analysis of CySeMoL models using ProbLog. For this implementation, we have constructed two software programs, which we describe next.

We have structured this chapter according to these two aforementioned programs, which are the following:

- The iEaat Parser, which transforms CySeMoL models into PVA and/or PVAI models.
- The analysis generator, which transforms a set of PVA and PVAI models into a ProbLog model, which is in turn transformed to a ProbLog program.

CySeMoL models are stored in EAAT instance `iEaat` files. These files contain the model containing specification of the vulnerability analysis and the model representing a concrete network architecture. The iEaat parser is able to extract these models from `iEaat` files, and transform them into a PVA or a PVAI model respectively. We will examine its implementation design and mode of operation in section 5.3.

The analysis generator program orchestrates the execution of model transformation scripts, with the goal of transforming PVA and PVAI models into a ProbLog program. This process consists of two parts, the model-to-model transformation of PVA and PVAI models into a ProbLog model, and the model-to-text transformation of ProbLog models into ProbLog programs. The details of the analysis generator implementation will be discussed in section 5.4.

5.2 Used Software

In the following sections, we use a variety of technologies for the implementation of our software programs. Before diving into the design of those programs, we will first describe these technologies. In order to provide a better insight into the state-of-the-art at the time of writing, we list the exact versions of each program or technology.

Eclipse Epsilon 1.3 The implementation of the Eclipse Epsilon framework. The framework provides implementations and APIs for Ecore, XMI, ETL, EGL and EOL. We use Eclipse Epsilon as the engine for our MDE transformation tasks. For more details on Eclipse Epsilon, see section 2.6.

Kotlin 1.0.3 Kotlin is an object-oriented open source programming language, developed by Jetbrains[42], which targets the Java Virtual Machine. We list some important features of the language below. For a more complete overview, we refer to the Kotlin website[42].

The language aims to be concise, reducing the amount of required boilerplate code to a minimum. In addition, it provides all the object-oriented features from Java 8 such as classes, interfaces, and traits. Kotlin code is able to call arbitrary Java code, and vice-versa, the generated Kotlin classes can be used in Java code.

Kotlin uses type inference to provide null-safety. At any point in the code, it is unmistakably clear whether a value can be null or not. The language provides specialized constructs which allow concise null-safe code to be written. Two of such constructs are:

- The null-safe invocation operator	`?.`
- The elvis operator	`?:`

The null-safe invocation operator specifies an operation on a potentially null variable, which is only executed if the variable is not null. The return type of the operator remains potentially null, which allows a chain of null-safe invocations to be specified. The elvis operator takes two arguments `A?` and `B`, and returns `A` if it is not null, and returns `B` otherwise.

It is possible to add methods to existing classes (i.e. the 'host' class) by defining *extension functions* (with similar semantics as the Epsilon Object Language, see section 2.6.2). When such extension functions are defined, they can be invoked on any instance of their host classes.

Lambda-functions, and functional idioms are supported, and have a prominent role in the language. For instance, the language provides (non-stream) `map`, and `filter` methods on `List` instances using a combination of *extension functions* and *lambda functions*.

Apache Maven 3.3.3 Apache Maven (usually abbreviated to just 'Maven') is a build automation tool aimed at Java projects. The build features provided by Maven are similar to the Ant program, which is also developed by Apache. Maven is a command-line tool, which reads a *Project Object Model* (pom) file. Such a `pom`-file is specified using the XML format, and contains the information required to automatically compile and package a software application.

The specification is based on the principle of *convention before configuration*, which means that only options which deviate from the defaults need to be specified. One of the assumptions made by Maven is on the folder structure of its projects. For instance, Maven expects Java source files to reside in the '`<project>/src/main/java`' folder, unless specified otherwise.

Moreover, Maven provides transitive dependency management, which allows Java projects to easily include a wide range of libraries from the central Maven repository. We use Maven to create a self-contained software package, and to automatically include all required dependencies in our project.

ProbLog 2.1.0.15.dev2 We use the second development version of ProbLog in order to perform probabilistic inference on the probabilistic logic programs which we generate. This specific version of ProbLog fixes a bug where the 'explain' feature of ProbLog did not function correctly. We run ProbLog under Linux, for the reason that the SDD library of ProbLog is only available under that operating system. In section 2.7, we have extensively discussed ProbLog and its features, and for more details on ProbLog we refer to that section.

CySeMoL 2.3 and EAAT v1.0.0 For our research, we have used version 2.3 of CySeMoL, in conjunction with build 2015025271427 of EAAT. This is the most recent version of P^{2}CySeMoL at the time of writing. We thoroughly explain the use cases and operation of CySeMoL in section 2.3. We used the *object modeller* from this version of EAAT to create the initial test cases for our research, as well as the example listed in figure 2.6. We have verified that our probabilistic vulnerability analysis framework is able to transform this version of the CySeMoL class model to the PVA model.

5.3 The iEaat Parser

5.3.1 Introduction

The iEaat Parser is the first tool which we have developed for our research. It was developed with the goal of transforming arbitrary CySeMoL (class and object) models in the `iEaat` format to our own PVA and PVAI models. The iEaat parser is able to transform the vulnerability analysis specification of CySeMoL (PVA), as well as network infrastructure instances (PVAI).

In this section, we will describe how we implemented our parser, and the methods used for the transformation. The software program itself is written in Kotlin, and is compiled into a single JAR file for distribution. To obtain the model classes using which we construct our PVA and PVAI models, we use EMF to generate model classes for our Ecore models. We package these model classes inside a `JAR` file, and include this file on the classpath of our transformation application.

5.3.2 Dissecting iEaat files

Recall from section 4.2 that the `iEaat` file format is a zip file, containing two models:

1. The Probabilistic Relation Model (PRM), which specifies the network components which can be part of a network infrastructure, and how the likelihood of compromise by an attacker can be inferred.

2. The Entity Object Model (EOM), which contains a set of network component instances, forming a network infrastructure.

We use the `ZipFile` class provided by the *Apache Commons Compress* library to decompress the the `iEaat file`. The `ZipFile` implementation is used to search for the PRM and EOM files, and to open up an input stream for each file. This allow us to read the contents of these compressed files without decompressing the entire `iEaat` file to disk.

In section 4.4 we have introduced our approach to transforming EOM models to our PVAI model. Our model transformation uses the SAX-parser from the Java standard library, which requires an input stream to an XML source for its operation. From the `iEaat` file we have already obtained such a stream, hence we are able to directly pass the EOM input stream to the EOM SAX-parser.

However, the PRM input stream contains a serialized Java object, which needs to be deserialized using the Java API. The Java class which provides this API is the `ObjectInputStream` class. This class is able to deserialize arbitary classes which are on the classpath of a running Java application. Consequently, we need to ensure that all Java classes stored in the PRM file exist on the classpath before we are able to employ the `ObjectInputStream` class To ensure that this is the case, we included all JAR libraries from the 'plugins' folder of EAAT into our own classpath. Using this method, we were able to deserialize the class model contained in the PRM file.

However, the method of including all EAAT dependencies requires the inclusion of over 200 JAR files which unnecessarily slows down the compilation process of the `iEaat` Parser. We reduced the amount of dependencies by removing JAR files one by one, and verifying whether the class model could still be correctly deserialized. Furthermore, we have created empty *mock* classes for some dependencies. For example, the `OCLClass` class also stores view information, and holds a reference to the `Rectangle` class from the (platform-dependent) Eclipse SWT graphics library. By loading these mock classes before importing the other classes from the JAR files, we are able to exclude most of the Eclipse graphics classes (and therefore their dependencies) stored in the PRM.

To summarize, by defining the following mock classes and interfaces:

- org.eclipse.core.runtime.IAdaptable interface
- org.eclipse.draw2d.geometry.Translatable interface
- org.eclipse.draw2d.geometry.Rectangle class
- org.eclipse.swt.graphics.RGB class

we have reduced our dependencies to the following 8 JAR files from EAAT and Eclipse:

- AbstractModellerPrm_1.0.0.201505271427
- ics.eaat.abstractmodeller_1.0.0.201505271427
- ics.eaat.abstractnode_1.0.0.201505271427
- ics.eaat.concretemodeller.main_1.0.0.201505271427
- ics.eaat.pivot_1.1.0.201505271427
- ics.eaat.property_1.0.0.201505271427
- org.eclipse.ocl.examples.pivot_3.5.0.201411241414
- org.eclipse.ui.views_3.7.0.v20140408-0703

The result of our efforts, is that we are able to obtain a deserialized version of the class model, which appears to be of the `ClassModel` type from the `ics.eaat.abstractmodeller.newmodel` package. This instance is sufficient to perform a complete transformation to our PVA model, and ultimately allows us to reproduce the vulnerability analysis of CySeMoL using ProbLog (see also section 5.4 and chapter 6). Furthermore, we have succeeded in excluding the dependencies on platform-dependent libraries, most notably the Eclipse SWT library.

5.3.3 The PRM contents

Using the debugger of our IDE (IntelliJ 15.0.6), we have analysed the structure of the newly obtained `ClassModel` instance. This instance provides us with access to the template, class and view definitions from the `iEaat` file. We only require access to the classes and the templates, which can be obtained from the class model as lists of `OCLClass` and `MetaTemplate` objects respectively.

Classes An `OCLClass` instance models an EAAT class, and is used to store the definition of the attack steps, defences, and assets of CySeMoL. Each `OCLClass` is allowed to have one single parent. This property is used by CySeMoL to identify classes as either an asset, defence, or as an attack step, by specifying the respective class as a parent.

EAAT allows the definition of operations for classes, which are stored in the `OCLClass` instance as well. Each operation stores its name, and body, and return type. For our purposes, we only require the use of the name and body of the operations, which are both stored as String instances.

`OCLClass` instances can be associated with each other by means of references. These references are bidirectional, and support different multiplicities and labels for each direction. One of the connected `OCLClass` instances is designated as the parent. References can also be ‘derived’, such references are defined a P^2AMF expression which specifies a navigation path from the parent of the reference by means of other existing references. The `OCLClass` target instance which is found at the end of the relation is stored within the reference as well.

Templates The `MetaTemplate` class stores an EAAT template definition. In its 'children' attribute it contains `TemplateObject` instances, which correspond to an `OCLClass` which will be instantiated for each instance of its owning `MetaTemplate` class. Additionally, the class defines a number of 'ports', which can be used to connect the type instantiation to other types. This port object points to a reference between two `OCLClass` instances.. When two `TemplateObject` instances are associated with each other, they are associated by their ports. For bookkeeping purposes, an association object is instantiated, which is referenced from the involved ports of these `TemplateObject` instances.

Templates are also able to be associated with each other. This is also achieved using ports, but these ports are stored in the `MetaTemplate` instances themselves instead of in the `TemplateObject` instances. The ports map a `TemplateObject` instance from the template to an instance of the `OCLCLass` class.

Finally, templates allow to visually display interesting attributes of their contents in the object modeller from EAAT. These attributes are modelled through the 'attributes' property of the `MetaTemplate` objects, and directly map onto `OCLClass` properties. In CySeMoL, template attributes are leveraged to display the results of the vulnerability analysis to the user. This is done by showing the estimated values of the success likelihood of attack steps within a template.

5.3.4 PRM data transformation

We will now discuss the process which we employ to extract the information from the PRM into a PVA model. We generate the data classes (everything except for the P^{2}AMF code) from the PRM according to the order specified in figure 4.1. The transformation process begins with the creation of an empty model container in the form of a `PVAContainer` instance, after which all other objects are created according to the dependency order mentioned earlier. The PVA model (chapter 3) requires that all instances are uniquely identifiable by a UUID. During the transformation of the PRM to the PVA model, we generate such UUIDs using the `java.util.UUID` class from the Java standard library.

In the following paragraphs, we will discuss the construction of the components of the PVA model in the following order:

1. Concrete Types
2. Templates
3. Special relations of assets and attack steps
4. Probabilistic properties
5. Virtual Connections

Concrete Types At the beginning of the transformation process, all classes from the PRM are partitioned by the name of their parent class. This name is obtained from the `OCLClass` objects. The resulting partitioning consists of the following partitions:

a) Assets

b) Attack steps

c) Defences

d) The attacker

e) Orphaned classes

As indicated by item e), we are unable to identify every OCLClass instance this way. This final partition contains two orphaned OCLClass objects: BypassDetectionSystems and NetworkVulnerabilityScanner. From the signatures of their methods in the PRM, we derived that these classes should be instances of an attack step and a defence respectively. We work around this issue by manually detecting these specific classes, and creating the corresponding types accordingly. As a consequence, these classes are indistinguishable from regular classes from the perspective of the PVA model.

For each item in the identifiable groups, an instance of either the AssetType, AttackStepType or DefenceType class is created. The name field of these instances is populated using the name in the OCLClass instance. All created instances are added to the model container as well, as we will search through the model container to retrieve previously generated instances later. While creating these instances, we already generate empty ProbabilisticProperty instances, as we need to obtain a reference to these instances from the templates later in the transformation process.

After all concrete types have been instantiated, we are able to transform all their references. From this point, we are able to retrieve any concrete type from the original model by searching through the model container. We proceed by collecting all reference objects from all OCLClass instances.

These reference objects store both their origin, and their target. By searching through the model container, we obtain previously generated ConcreteType instances for the source and target of each reference. All references in CySeMoL are bi-directional, however, in our PVA model we use two uni-directional relations instead. In order to retain the bi-directional capabilities of the relations, each created relation stores both its label and the name of its relation. The label will be used to navigate the PVA model later.

Furthermore, we determine at this point whether a relation should be virtual or not, by checking whether the derivedBody field of the reference has been populated. We will fill the path relation of the VirtualConnection instances at the very end of the transformation process. We choose to create these objects at this point, as we need to be able to retrieve their names for path traversals in other transformation steps.

Templates After creating the concrete types and their connections, we are able to construct templates. We do not use the template list directly, but obtain the defined templates by querying the OCLClass instances. This is achieved by retrieving the instantiations of these classes (by following the instance relation), and then obtaining the parent of these instantiations, which corresponds with a template. By removing duplicate templates, we are left with a set of meaningful templates. Unused or empty templates are not transformed. After obtaining a list of all unique template definitions (in the form of MetaTemplate instances) stored in the PRM, we filter out the attacker, as the attacker is modelled implicitly in the PVA model. For each template, we create an instance of the TemplateDefinition class from our PVA model, and assign the name of the template to it.

Next, from the PRM templates, we create TypeInstantiation instances for all instance types which have been defined for those templates. These instances are stored in the getTemplateChildren method, which returns a list of TemplateObject instances. The TemplateObject instance holds a reference to the OCLClass instance it is an instantiation for. We retrieve the ConcreteType counterpart for this OCLClass by searching in the PVA model container. The retrieved concrete type is assigned as the concretization of the newly created type instantiation. For an overview of the relations between the involved PRM and PVA classes in this transformation, see figure 5.1.

After we have generated all type instantiation instances for this template, we generate the connections between those instantiations. Here we make a distinction between connections within a template (internal) and between templates (external). Every TemplateObject instance contains a list of its internal associations. We use this list to generate the internal connections for the current template definition.

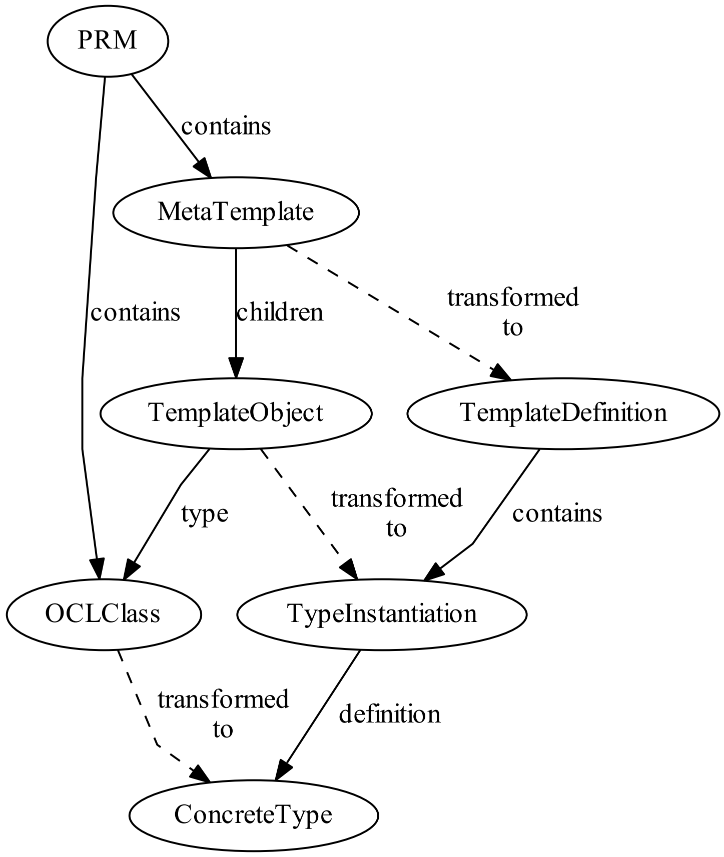


Figure 5.1: An overview of the transformation relationships of PVA template definitions, type instantiations, concrete types, and their PRM counterparts.

An association is bi-directional but is defined using a source and a target, the bi-directionality lies in the fact that both the source and target of an association refer to the same association object. To determine whether the current `TemplateObject` is the source or the target, we examine these fields of the association first. After this point, we have a reference to two `TemplateObject` instances, for which we look up the `TypeInstantiation` counterpart in the PVA model container. Using this information, we are able to create a `InstantiationConnection` instance, and populate all its fields. Since we know which type instantiations are connected, as well as the relevant association, we are able to determine the concrete connection which resembles this instantiation connection. We achieve this by matching the name and labels of this connection with the connections between the `ConcreteType` definitions of the type instantiations.

In addition to the internal connections, we also search for the concretization of the external connections. We obtain the external connections in the PRM from the `connectionPorts` list. This list contains all potential connections of classes in the template, both internal and external. Which we first remove the internal connections, by comparing the references of the ports to the port objects found in the `associations` list used for the creation of the `InstantiationConnection` instances. From the references stored in the remaining ports, we are able to search through the model container for any `ConcreteConnection` instance which matches this reference.

Since we are able to reference all type instantiations and their connections from this point, we are able to transform the template attributes. We retrieve the list of attributes from the MetaTemplate instance which are stored in its 'attributes' property. We designed our PVA model to only support attributes reflecting either the success likelihood of attack steps or whether defences are functioning. For this reason, we filter the attributes for these properties by checking whether their concrete target is one of such properties. We distinguish the type of an attribute through the name of its target. By the definition of inheritance in the PRM, all attack step attributes have the name 'likelihood', and all defence functioning attributes have the name 'functioning', which have been inherited from the AttackStep and Defense classes respectively. Each attribute stores the OCLClass owner of the attribute, therefore we are able to identify the AttackStepType or DefenceType instance which owns the property, and add a reference to that property to the TemplateDefinition attribute list.

This concludes the transformation of templates definitions from the PRM model to our PVA model. When each template definition has been completed, it is added to the template types bin of the model container.

Special relations This step is performed after the transformation of templates, but may be performed earlier if desired, as this step only depends on the presence of all concrete types. The transformation procedure starts by determining which OCLClass instances map to assets, and which instances map to attack steps. The method for this determination is the same as for the transformation of the concrete types. Again, we use the superclass of the OCLClass objects to determine whether a class is either an asset an attack step, or a defence.

For attack steps, the AttackStepType.target relation is populated by finding all attack steps which reference the OCLClass associated with each asset. The contents of AssetType.defendedBy relation in turn are formed by those defences which have a reference to the OCLClass associated with each asset.

The AttackStepType holds a enables reference to all the attack steps it enables. In the PRM, this information is stored in the getPaths method of the corresponding OCLClass of the attack step. For every attack step, we parse the paths stored in the getPaths method (see section 5.3.5, where we show how this is performed), and resolve those paths within the PVA model by navigating a chain of ConcreteConnection instances. This way we immediately obtain an instance of the AttackStepType class without having to resort to additional lookups. The referenced attack steps are added to the enables relation.

Probabilistic properties The PRM of CySeMoL defines probabilistic properties which specify the derivation of the success likelihood of attack steps and the probability that defences are functioning. In the PVA model, the AttackStepType and DefenceType have been designed to support such probabilistic properties. From the PRM, the P^{2}AMF bodies of the operations containing the derivation of the probabilistic properties are stored in the OCLClass objects. By parsing these derivations, we are able to generate a ProbabilisticCode instances by the process described in section 5.3.5. This instance is wrapped in the implementation field of a newly created ProbabilisticProperty instance, which is then assigned to the attack step or defence type corresponding to the owning OCLClass instance from the PRM.

Virtual connections At the very end of the PRM transformation process we will tend to the transformation of the derived edges, which are mapped to the `VirtualConnection` class in the PVA model. The process of generating virtual connections is similar to the generation of concrete connections. During the construction of the concrete types, we have already created `VirtualConnection` instances (see the part on concrete types at the beginning of this section). During the creation of these connection instances, we have cached their PRM counterparts in the form of a `Reference` object, which we look up to obtain the derivation body. This derivation specifies the navigational path which defines this virtual connection. By the procedure specified in section 5.3.5, we obtain the concrete relations required to travel to the goal of a virtual connection. We reproduce this traversal within the PVA model, in order to retrieve a list of `ConcreteType` instances. The list of intermediate connections are added to the `path` list of our current `VirtualConnection` instance.

5.3.5 P^{2}AMF transformation

We will now tend to the implementation of the P^{2}AMF parsing routines. For their context and general design, see section 4.3.3. As mentioned in section 4.3.3, we have designed three P^{2}AMF parsers. We have integrated all parsers within the PRM transformation code.

We will begin by describing the design of our large P^{2}AMF parser, which is used to transform the `AttackStep.isAccessible` and `Defense.isFunctioning` from P^{2}AMF to our own model. The parsing of such methods is invoked when the PRM transformer reaches an instance of the `AttackStep` or `Defense` class. First, the program is parsed by an ANTLR4 parser generated according to the grammar as listed in appendix B.2, this gives us an abstract syntax tree of the P^{2}AMF program. The parser takes care of some P^{2}AMF statements which are redundant in our model. Specifically, we remove references to 'self', and replace probability distributions nested within a Bernoulli distribution with just the wrapped distribution. This construct is used in CySeMoL to cast a probability distribution into a distribution which generates boolean values. As our approach does not use sampling, we do not need to perform this operation.

Next, an instance of the `P2AMFBuilder` class is created, and its `visitProgram` method is invoked on the AST generated by ANTLR. This invokes the transformation process, ultimately resulting in an instance of the `ProbabilisticCode` class which stores the converted program according to the semantics of our PVA model.

In order to explain the transformation dependencies, we define the $\mathscr{T}$ function, which denotes the transformation of its argument from P^{2}AMF to an instance of the `ProbabilisticExpression` class. In the following text, we will use this function to refer to the transformed counterpart of P^{2}AMF statements.

Most of the AST nodes from the P^{2}AMF parser output are transformed by creating a relevant instance of a class from the `probcode` package of the PVA model, and populating this instance with the the output of the transformation of the child nodes. For instance, the P^{2}AMF statement "χ **and** ξ" is transformed to an instance of the `probcode.And` class, with $\{\mathscr{T}(\chi), \mathscr{T}(\xi)\}$ as its children. When we ignore code optimizations, which are described in section 5.3.6, the transformation of most other statements from P^{2}AMF is performed similarly, as shown in table 5.1. The non-trivial transformations are the referential statements and the linear distribution.

The linear probability distribution has been described in section 2.3.7, and is specified using a list of x-coordinate values, and a list of y-coordinate values. When transforming a linear probability distribution definition, the lists are zipped together into pairs of coordinate values, forming a list of coordinates. This list of coordinates is converted to a list of `ApproximationPoint` instances, which is in turn wrapped in a new `LinearDistribution` instance.

Table 5.1: An overview of the straightforward transformations of P^{2}AMF to PVA. $\mathscr{A}$ is a reference to the time spent by the attacker.

P^{2}AMF statement	*Transformation result*
χ **and** ξ	`And`, children: $\{\mathscr{T}(\chi), \mathscr{T}(\xi)\}$
χ **or** ξ	`Or`, children: $\{\mathscr{T}(\chi), \mathscr{T}(\xi)\}$
not χ	`Not`, child: $\mathscr{T}(\chi)$
if χ **then** ξ **else** ψ	`IfThenElse`, condition: $\mathscr{T}(\chi)$, true: $\mathscr{T}(\xi)$, true: $\mathscr{T}(\psi)$
true	`Constant`, value: 1.0
false	`Constant`, value: 0.0
c, $c \in [0,1] \subset \mathbb{R}$	`Constant`, value: c
bernoulli(c), $c \in [0,1] \subset \mathbb{R}$	`Bernoulli`, probability: c
exp$(l, \mathscr{A})$, $l \in \mathbb{R}$	`ExponentialCDF`, lambda: l
normal$(m, s, \mathscr{A})$, $m, s \in \mathbb{R}$	`NormalPDF`, mu: m, sigma: s
lognormal$(m, s, \mathscr{A})$, $m, s \in \mathbb{R}$	`LogNormalCDF`, mu: m, sigma: s
gamma$(a, b, \mathscr{A})$, $m, s \in \mathbb{R}$	`GammaCDF`, alpha: a, beta: b

Table 5.2: Conversion of P^{2}AMF referential statements.

P^{2}AMF format	Generated instance type
`visited → intersection(<reference>) → notEmpty()`	`AttackStepSucceeded`
`defenseAvailable(<reference> → asSet())`	`DefenceOperational`
`<reference> → notEmpty()`	`ConnectionAvailable`

P^{2}CySeMoL code can also contain structured referential statements: a sequence of relations which can be navigated to arrive at an attack step, defence, or asset instance. We encounter this type of statements in the `isAccessible` and `isFunctioning` implementations, as well as in the `getPaths` methods and particularly in the specification of derived edges. For the probabilistic P^{2}AMF properties, we have defined the `ReferencingOperand` interface. Classes which implement this interface own a 'reference chain', a sequence of `ConcreteConnection` instances, which are used to model the sequence of relations. Depending on the form of the P^{2}AMF code, and the target of the reference chain, we generate one of the following instances:

- `DefenceOperational`, which queries whether a defence is functioning.
- `AttackStepSucceeded`, which queries whether an attack step has been achieved.
- `ConnectionAvailable`, which queries whether an asset is reachable by the provided path.

The P^{2}AMF patterns are converted as shown in table 5.2. The reference chain is obtained from the AST, which contains a list of navigable labels. The labels are used to iteratively obtain the intermediate `ConcreteConnection` instances and ultimately arrive at the target `ConcreteType` instance. The pattern for transforming reachability might reference other types, however, this pattern is used in P^{2}CySeMoL to retrieve assets only.

Table 5.3: An overview of the P^{2}AMF optimizations applied during the PRM parsing process.

Source pattern	*Optimized into*
if not c **then** t **else** f	**if** c **then** f **else** t
if c **then** x **else false**	c **and** x
if c **then true else false**	c
if c **then false else true**	**not** c
if not c **then false else true**	c

The `getPaths` and the derived connection bodies use a slightly different format of P^{2}AMF code to specify their contents. To this end, we developed two separate ANTLR4 scripts (listed in appendices B.3 and B.4) to be able to correctly parse those pieces of code. On an high level, the `getPaths` bodies consist of a chain of unions on individual attack steps, which results in a set of reachable attack steps. This formatting allows us to parse the `getPaths` methods, and obtain a list of `AttackStepType` objects, by looking up the reference chains in a similar way as described before. The resulting set is used to populated the `AttackStepType.enables` edge. The derived edges use an even simpler format, as they only contain a single navigable expression which points at some `OCLClass` instance. Using the path obtained from the AST generated by the ANTLR4 parser, we again obtain both the reference chain and the target, which we store in the `VirtualConnection.path` relation.

5.3.6 P^{2}AMF optimizations

During parsing, we apply some optimizations of the P^{2}AMF on-the-fly. We noticed some patterns in the P^{2}AMF code blocks of P^{2}CySeMoL, which had room for easy improvements. These optimizations have been implemented in the P^{2}AMF walker code from the PRM transformer which walks over the AST generated by ANTLR.

We list the individual optimizations in table 5.3. The resulting code after applying these optimizations requires fewer operators, and results in a smaller PVA model. These optimizations resulted in improvements in the speed of both our model transformation and the time required for performing vulnerability analysis with ProbLog.

5.3.7 EOM parser implementation

The transformation of the EOM input model from CySeMoL differs from the transformation to the PVA model. A major difference is that we have access to the plain-text XML description, which is also used as an input by CySeMoL. It turns out that this is more important than expected, as the deserialization of EOM files has some bugs in CySeMoL, which are caused by the structure of the EOM format. These bugs are described in more detail in section 6.5.4.

The EOM parser has been implemented as a SAX parser[61]. It consists of a single class, which extends the `DefaultHandler` class from the `org.xml.sax.helpers` package of the Java 8 standard library. By using the SAX parser API, our parser functions as a stateful listener. This method uses a notification architecture, in which our listener is notified of XML tags by the SAX parsing back-end, in the order that they are encountered. To hide the statefulness from external users of our class, we provide a single `parse` method. The `parse` method takes the dissected `iEaat` parts as an argument, and returns an instance of the PVAI model container. Our implementation assumes that the XML structure of the EOM file conforms to the XMI serialization of the model shown in figure 4.2. By making this assumption, we are able to correctly determine the context of data when the SAX back-end signals that a new XML tag has been reached.

The parser is able to handle the following four cases of tags, which we will describe in order:

1. The root tag, which contains the XMI metadata. This information is discarded by our parser, as it is not used in the PVAI model.

2. A `objects` tag, which indicates the definition of a template instance.

3. A `evidenceAttributes` tag, which specifies evidence for the current context.

4. A `associations` tag, indicating the definition of a relation between templates.

Template Instances The `objects` tag contains the information which is used to specify instances of templates. In P^{2}CySeMoL, the attacker is also a template, whereas we have defined a special class in the PVAI model. Therefore, the first action of the instance definition handling code is to detect whether the encountered tag specifies an attacker, or a template instance. When we encounter an attacker, we create an instance of the `Attacker` class in the PVAI container, and obtain the name of the attacker from the `objects` tag.

The detection of template instances is more involved. Our current specification of the PVAI model does not support two template instances with the same name, whereas the EOM format allows this. It is unclear whether CySeMoL supports this feature, as it is possible to serialize models with multiple template instances with the same name. We work around this limitation of our PVAI model by detecting duplicate template names, and generating a unique name by appending an identifier to the template instance name. When all potential problems have been either avoided or corrected, we generate an instance of the `TemplateInstance` class, and set our current scope to that instance. Additionally, we cache the identifier of that template as specified in the EOM file, which allows us to efficiently retrieve `TemplateInstance` objects based on their EOM identifier when we need to resolve associations later on. The `definition` field of the generated `TemplateInstance` object is populated by searching the PVA model for the corresponding `TemplateDefinition` instance. This search is performed using the name provided in the `metaConcept` attribute of the `objects` tag.

Evidence Evidence is specified in a `evidenceAttributes` tag, within the context of a template instance. The EOM parser stores the current context, which allows us to retrieve the context when the SAX parser notifies the EOM parser of an encountered `evidenceAttributes` tag. First, the evidence name is generated based on a combination of the property it refers to, and the name of its owner. Conjointly, the boolean value of the evidence is extracted from the tag attribute.

After the gathering of initial information for evidence, the EOM parser searches for type instantiations within the owning template definition to which the evidence can be applied. This search is based on the name of the type instantiations. Due to a bug in EAAT, it is possible to specify evidence for multiple type instantiations with the same name. We work around this bug by applying the evidence to only one of the multiple instantiations, and by warning the user of this bug. Depending on the type of the target instantiation which can be either an `AttackStepType`, or a `DefenceType`, the corresponding (probabilistic) property is retrieved and stored as the target of the evidence. Finally, the now completed `Evidence` object is assigned to the current `TemplateInstance` context.

Associations Finally, the EOM parser processes the associations between template instances, as defined using the `associations` tags. For our implementation, we assume that all other tags (especially the relevant `objects` tags) have been processed at this point. The `associations` tag contains identifiers which refer to earlier defined `objects` tags. The target of these references is either the attacker, or one of the template instances. Recall from the previous paragraphs that the EOM parser has cached the identifiers. We use this cache to look up the created instances, and to check whether we are dealing with any references to the attacker.

When the association involves the attacker, we generate an `EntryPoint` instance. The entry point requires a target and an attack step for that target, which is looked up from the PRM. We have already retrieved the target from the identifier cache, which leaves the EOM parser with the task of finding the correct attack step. Within the PRM, we filter the associations from the attacker template to the target template on the provided association in the EOM file. Note that in practice, the label as specified in the EOM is not unique and can refer to multiple entry points simultaneously. This case is handled incorrectly by EAAT as described in section 6.5.4. When the EOM parser encounters this erroneous case, the algorithm selects the first match for the attack step, and the user is warned of the potential problems and notified which concrete entry point has been selected.

The other type of association is between template instances. From the identifiers from the `associations` tag, we are able to retrieve the relevant `TemplateInstance` objects. However, an association between templates maps to a concrete bidirectional connection between two type instantiations, which the algorithm needs to look up from the PVA model. From the `definition` relations of the template instances, the EOM parser accesses the definitions of the templates. From the external connections of the source template, the potential type instantiation targets of the association are obtained. By filtering on the relation name, only a single association is left (provided that the PRM is not ambiguous), which leaves a single potential `ConcreteConnection`. Following this procedure, we have obtained all required information referenced by the `ConnectionTemplate` instance, which models associations between template instances. The EOM parser creates and populates such an instance, and adds the association to the source template. The aforementioned procedure is repeated with the source and target templates switched, in order to create a bidirectional relation.

5.3.8 Interoperability

In order to exchange our constructed PVA and PVAI models, we serialize our generated Java data structure using XMI. The XMI format is independent of Java, and therefore provides better interoperability than using Java object serialization using the `ObjectOutputStream` class.

The XMI serialization code recognizes the model classes (as they implement an EMF-specific interface), and is able to easily serialize the PVA model. However, for the PVAI model, we noticed that simply serializing the model resulted in both the PVA *and* the PVAI model being serialized. Our intention being that multiple PVAI models could reference a single PVA model, this was not ideal. We resolved this by explicitly unloading the PVA model before serializing the PVAI model, which causes the generation of proxy references where the PVAI model classes reference PVA classes. The XMI serializer serializes these proxy references to an external reference in the generated XMI file. Using Eclipse Epsilon in the next transformation stage, we were able to successfully load and read the serialized PVA and PVAI models.

5.4 The Analysis Generator

5.4.1 Introduction

Using the data from the PRM contained in the PVA and PVAI models, we want to reproduce the analysis of CySeMoL by means of the inference of probabilities within a ProbLog program. In this section, we will describe our design of the vulnerability analysis of a combination of a PVAI and PVA models, specified in ProbLog. We implement the generation of the ProbLog program using model-to-model transformations and model-to-text transformations, as shown in figure 3.1. The execution of the transformation scripts has to be orchestrated. To this end, we have developed the 'analysis generator' program, which is able to transform the PVA and PVAI models to ProbLog.

The PVA and PVAI models roughly consist of data and definitions of probabilistic derivations. How these probabilistic inference is executed is left open (e.g. independent of ProbLog or CySeMoL), which allows for multiple potential implementations. In this section, we will describe our approach to the implementation of these derivations using ProbLog. The construction of our probabilistic analysis using ProbLog can be separated into two parts. On one hand, we have a dynamic part which depends on the information contained in the PVA and PVAI models, specified as a long list of facts. On the other hand, we have a static part of ProbLog code, which specifies how the requested probabilities are derived from the first part.

The ProbLog code generated by the transformation script which transforms the PVA model uses many auxiliary ProbLog rules. These rules are the same in every program, as they are used to capture most of the model and analysis semantics. The auxiliary ProbLog code is kept separate from the transformation scripts, and is only added to the generated ProbLog program after all transformation steps have been completed.

In the analysis generator program, the ProbLog code is stored as a Java resource, which ensures that the code is packaged inside the program JAR file on deployment. As the PVA and PVAI models can be transformed to ProbLog in any order, the analysis generator supports partial and complete transformations (see also section 5.4.7). On a complete transformation, the program injects the static code into the output file, producing a legal executable ProbLog program. Summarizing, we inject three parts of static code:

1. The auxiliary ProbLog code, which forms the base of our vulnerability analysis (section 5.4.2).
2. The queries which instruct ProbLog to infer the required probabilities (section 5.4.3).
3. Implementations for the probability distributions used in the vulnerability analysis (section 5.4.4).

We will first describe the implementation details of the three static code blocks. After these sections, we follow up with the dynamic code transformation in section 5.4.5. Note that the static code references components from the dynamic code, and vice-versa, which means that it may be required to read all four sections. Next, we describe how the model-to-text transformation of ProbLog models to ProbLog code is implemented. Finally, we will describe the design of the 'analysis generator' program, and how the analysis results from ProbLog are presented to the user.

5.4.2 ProbLog Analysis base implementation

One of the main purposes of the base ProbLog code is the coupling between types and instances of types. From the specification of the existence of template instances using ProbLog facts, we let ProbLog automatically derive the instances of concrete classes which should be instantiated. The information on how this should be done is stored in the type instantiations of the templates, which will be stored as facts in the ProbLog program. Moreover, we assume the existence of all attack steps which target an asset.

To model the existence of instances in ProbLog, we have defined the `instance/4` predicate. The first argument of the predicate is an atom which represents the type of the instance. We have defined the following atoms:

1. `attackStepType`, for attack step instances.
2. `defenceType`, for defence instances.
3. `assetType`, for asset instances.

The instance definitions for the assets and defences have the following arguments:

1. The instantiation name which identifies multiple instantiations of the same type.
2. The name of the type.
3. The name of the template instance which defines the instantiation of this instance.

In other words, an instance definition is derived from the existence of type instantiation for some template instance. For the attack steps however, we do not use instantiations, but depend on the `attackedBy/2` predicate. For each asset instance, we generate instance definitions for all attack steps which target the type of the asset instance. As a consequence, we do not store the instantiation name of the attack step in the `instance/4` predicate, but the instantiation name of the asset which is targeted by the derived attack step instance. To demonstrate this feature, we list a few derivations:

```
instance(assetType,applicationServer,applicationServer,web_Server).
instance(attackStepType,applicationServer,floodDoS,web_Server).
instance(defenceType,secretRoaming,secretRoaming,web_Server).
```

Similar to the derivation of instances of types, we have to derive instances of connections. In this case, we have specified rules which derive a `concreteConnectionInstance/4` for each generated `instantiationConnection/4` fact for which a template instance exists. The information stored in both predicates is nearly the same, we first have a role name, then the source and target instantiations within a template, and finally the template name. The only difference is that the connection instantiation refers to its template definition, whereas the concrete connection instance refers to a template instance. As we have automatically constructed attack step instances, we also have to generate concrete relations for those instances. To achieve this, we do not resort to connection instantiations, but base the configuration of the concrete connection instance on the concrete connections between the attack step types and the assets which they attack.

In the base program, we have defined static rules which we use for the resolution of the virtual connections. The facts generated for the dynamic part or the program contain a list of tuples which form edges and nodes the paths. For the path resolution, we use the following four predicates:

1. `getAnyPath/3`, which navigates the connections from instance to instance.
2. `virtualConnectionInstance/3`, which allows the discovery and resolution of virtual connections when one exists.
3. `reverseVirtualConnectionInstance/3`, due to CySeMoL's bidirectional relations, it is sometimes required to resolve virtual connections backwards, which is reflected in this predicate.
4. `connectionExists/3`, used for the discovery of any kind of connection between instances.

The `getAnyPath/3` predicate defines a sequential search for instances which can be reached by a given path. The elements from the path are navigated until an endpoint is reached. The arguments of this predicate are the source instance, target instance, and the desired path. The path is specified as a list of label/type tuples, which specify both the label of the connection which should be navigated and the type of the target. By setting either the source or target as a variable, ProbLog can be used to resolve the reachable instances using the given path.

The `virtualConnectionInstance/3` and `reverseVirtualConnectionInstance/3` have the same role as the `concreteConnectionInstance/4` predicate, with the difference that the virtual predicates are defined on top of concrete connections and potentially also other virtual connections. The existence of a virtual path is proven by verifying whether the provided path for a virtual connection between two types can be navigated. To verify this navigation, the `getAnyPath/3` predicate is used. Reversed virtual connections are resolved by switching the source and target instances. This causes ProbLog to determine all source instances for which the target instances is reachable by the given path.

Finally, the `connectionExists/3` predicate is used to generalize over connections. The arguments of this predicate again consist of a relation label, source instance, and a target instance. The predicate can be proven when either a virtual or concrete connection can be proven. The `getAnyPath/3` rules use this predicate to resolve individual steps within a path. Consequently, virtual connections are allowed to refer to other virtual connections in their paths.

In the probabilistic code in the PVA model, we support checking whether some attack step has succeeded, whether some defence is functioning, or whether some asset exists. This feature is invoked by specifying a path to the respective instance which should be evaluated. In the static ProbLog code, we define auxiliary rules which are able to resolve the path and simultaneously evaluate the required property. To implement this feature, we define the following three predicates:

1. `checkAttackStepSucceeded/3`

2. `checkDefenceWorking/3`

3. `checkConnectionExists/3`

All predicates take the same arguments, a source instance, a target type, and a path to the target type. Their operation is also similar, first, the path is resolved to test which instance can be reached, and to verify its type. Additionally, the `checkAttackStepSucceeded/3` and `checkDefenceWorking/3` predicates require that their respective probabilistic property can be proven. This is achieved by invoking the dynamically generated predicates for this task:

1. `attackStepSucceeded/4`, which can be proven with the probability that the requested attack step succeeds.

2. `defenceIsFunctioning/1`, which can be proven with the probability that the requested defence instance is functioning.

By using these predicates, the dynamic code is able to verify the required properties using a single predicate instead of having to generate multiple predicates.

Finally, the dynamic code for attack steps requires the determination of the set of reachable attack steps. Before a probabilistic property is evaluated, the `somePathToAttackStep/4` predicate is evaluated to determine whether the current attack step instance is reachable. The reachability graph is built by starting from the entry points, and then using the `attackPrerequisite/2` facts whether all prerequisites for some attack step are reachable. For the entry points, we use the `entrypoint/3` fact, which is dynamically generated. For all other attack step instances, we require that at least one `attackPrerequisite/2` targets the current attack step instance, and that the `somePathToAttackStep/4` predicate can be proven for the source attack step instance.

Evidence in CySeMoL has different semantics than the observation of probabilistic events. In particular, evidence is CySeMoL simply overrides the outcome for the variable the evidence is defined on. This led to some problems in ProbLog, which does not allow the specification of evidence which conflicts with a non-probabilistic derivation. We solved this problem by introducing the `hardEvidenceTrue/1` and `hardEvidenceFalse/1` predicates. The argument of these predicates is an instance definition of an attack step or a defence.

These rules are used to override the (normally probabilistic) evaluation of the `attackStepSucceeded/1` and `defenceIsFunctioning/1` predicates. The overriding nature is achieved by specifying two new rules, which allow ProbLog to directly prove the two probabilistic success rules from the evidence predicates. Using this system, the evidence can be generated by the PVAI transformation script. The following example demonstrates such a generated fact:

```
templateInstance(web_Server, tApplicationServer).
hardEvidenceTrue(
  instance(defenceType, loadBalancer, loadBalancer, web_Server)
).
```

5.4.3 Querying the ProbLog program

In order to obtain the desired information from the ProbLog program, we have to generate queries which will retrieve this information. For our vulnerability analysis we are interested in the following probabilities:

1. The success probability for every attack step instance.
2. The likelihood that a defence is operational.

We have captured both properties in the `attackStepSucceeded/4` and `defenceIsFunctioning/1` predicates. Therefore, we use a query containing variables to let ProbLog determine the applicable attack steps and defences, and to infer the probability of the chosen predicates. Our queries have the following form:

```
query(attackStepSucceeded(_,_,_,_)).
query(defenceIsFunctioning(instance(defenceType,_,_,_))).
workdays(5).
```

Note the inclusion of the workdays parameter, which sets the amount of work days used in the analysis. We currently use this parameter to mimic the analysis of CySeMoL, which is why we include this predicate in the query code. When a different analysis approach is used, this code has to be adapted accordingly.

5.4.4 Probability Distribution implementations for ProbLog

P^{2}CySeMoL uses a set of probability distributions which it uses during for the generation of probabilities during its vulnerability analysis (see section 2.3.5). In order to reproduce this analysis, we have to provide implementations for all the probability distributions used by CySeMoL in ProbLog. Fortunately, ProbLog provides us with some auxiliary maths functions[15], which we can use to construct the probability density and cumulative probability functions.

We use the feature from ProbLog which supports dynamic probabilities for facts, given that these probabilities can be calculated when required. Provided, we have developed a function $f(x,\vec{p})$, which returns a probability value for a given value of x and a set of parameters $\vec{p}$, we can specify this in ProbLog as follows:

$$\texttt{Probability::F(}x\texttt{,}\vec{p}\texttt{) :- Probability is } f(x,\vec{p})\texttt{.}$$

If we are able to compute $f(x,\vec{p})$, ProbLog is able to prove `F(`x`,`$\vec{p}$`)` with probability $f(x,\vec{p})$.

As we are reproducing the P^{2}CySeMoL analysis, we have also generated auxiliary functions, which invoke the probability distribution implementations with the value of x which corresponds to the amount of work days. In practice, this means that for our analysis we use the value of $x = 5$, as P^{2}CySeMoL sets the amount of work days to 5 by default. The reason that we provide general implementations, is to abstract from this single value, and allow multiple values to be used. Nevertheless, it is possible to provide different implementations for these distributions, which do not depend on a static amount of work days.

Bernoulli distribution We have implemented the Bernoulli distribution using annotated disjunctions.

```
Output::bernoulli(X) :- Output is X.
```

Exponential cumulative density function The exponential cumulative density function has been implemented using the ProbLog built-in functions of e^x. This way, we are able to implement the following function:

$$\text{expCDF}(x, \lambda) = 1 - e^{-\lambda x} \qquad\qquad x > 0$$

Normal probability density function The normal probability density function can be implemented according to its definition:

$$\text{normalPDF}(x, \mu, \sigma) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Log-Normal cumulative density function The log-normal cumulative density function can be implemented according to its definition, using the build-in ProbLog function for the complementary error function (erfc):

$$\text{logNormalCDF}(x, \mu, \sigma) = \frac{1}{2}\text{erfc}\left(\frac{\log(x) - \mu}{\sigma\sqrt{2}}\right)$$

LogGamma function For some of the formulae used to calculate the exact probabilities from a distribution, we require an implementation of the *LogGamma* function. We use the following formula to approximate the value for the LogGamma function, as specified by Macleod[54] (constants have been rounded to 4 decimals for readability):

$$\log(Gamma(x)) \approx \log\left(\frac{0.1659 \cdot 10^{-6}}{x+7} + \frac{0.9935 \cdot 10^{-5}}{x+6} - \frac{0.1386}{x+5} + \frac{12.51}{x+4} - \frac{176.6}{x+3} + \frac{771.3}{x+2} - \frac{1259}{x+1} + \frac{676.5}{x} + 1.000\right) - 5.581 - x + \left(x - \frac{1}{2}\right) \cdot \log(x + 6.5)$$

The reason for taking the logarithm, is that it prevents potential arithmetic overflows caused by the Gamma function. For the final calculation, we raise the result of the logarithmic formula to the e-th power to obtain the normalized result.

The GammaCDF function The cumulative density function of the gamma distribution does not have a closed form notation using the functions provided by ProbLog. However, there exist approximation algorithms for the cumulative gamma distribution. The Apache Commons Math 3.6.1 library contains an implementation for the cumulative Gamma distribution in the `GammaDistribution.cumulative` method. This method uses the definition of the cumulative Gamma distribution in terms of the regularized gamma function $P(a, z)$ and the incomplete gamma function $\gamma(a, z)$:

$$\begin{aligned} D(x) &= P(k, \frac{x}{\theta}) \\ &= \frac{\gamma(k, \frac{x}{\theta})}{\Gamma(k)} \end{aligned}$$

the Java implementation approximates the logarithm of $\gamma(k, \frac{x}{\theta})$ using an infinite sum[80], and calculates $\log(\Gamma(k))$. These values are used to obtain a value for $\log(D(x))$, which is then raised to the e-th power to obtain a value for the cumulative density of the Gamma distribution.

We have copied this implementation to ProbLog, and use our own implementation of the log-Gamma function as described above. We choose an error bound of $\epsilon = 10^{-6}$ for our approximation, which should be accurate enough for the purpose of indicating the likelihood of compromise for assets.

Table 5.4: An overview of the trivial parts of the generated ProbLog code for the PVA and PVAI data. The top classes originate from the PVA model, whereas the bottom classes have been defined within for PVAI model.

Source	*Generated ProbLog template*
`ConcreteConnection`	`concreteConnected(relationName,source,target).`
`TemplateDefinition`	`templateDefinition(name).`
`AssetType`	`assetType(name).`
`AssetType.defendedBy`	`defendedBy(name).`
`DefenceType`	`defenceType(assetName,defenceName).`
`AttackStepType`	`attackStepType(name).`
`AttackStepType.target`	`attackedBy(assetName,attackStepName).`
`TypeInstantiation`	`assetInstantiation(name,typeName,templateName).`
	`defenceInstantiation(name,type,template).`
`AttackStepType.enables`	`attackPrerequisite(sourceName,targetName).`
`TemplateInstance`	`templateInstance(name,definitionName).`
`EntryPoint`	`entryPoint(targetName,attackStep,templateName).`

Linear approximation of probability distributions The authors of CySeMoL use the EAAT feature of approximating a probability distribution using a list of points. When evaluating the probability for that distribution, linear interpolation is used to determine probability values for points not included in the list. In ProbLog, we have implemented this feature using the `linearApproximate/2` predicate, which takes a value of x and an ordered list of points (sorted on the x-coordinate of the points), and can be proven with the probability of x according to the list of points.

The `linearApproximate` iterates over the list, and looks for a sequence of two points for which x lies between the x-coordinate values of the points. If this sequence cannot be found, the function returns a probability of 0. Otherwise, it linearly approximates the probability using the following formula:

$$p(x,(x_1,y_1),(x_2,y_2)) = y_1 + \frac{x-x_1}{x_2-x_1}\cdot(y_2-y_1) \qquad x\in[x_1,x_2)$$

the resulting value for $p(x)$ is returned as the probability of the `linearApproximate` predicate.

5.4.5 The ProbLog facts

In this section, we will focus on the representation of the data from the PVA and PVAI models in ProbLog. We have separated the generation from the PVA and PVAI models in such a way, that the PVAI models can be exchanged, as long as they refer to the same PVA model. For the trivial transformations, we list a template-based description of the generated ProbLog code in table 5.4. These trivial transformations consist of the generation of facts containing the information contained in the source class.

The first non-trivial transformations which we will discuss are the virtual connections. These connections consist of the information contained in a regular connection, and a path from the source to their target. Moreover, the virtual connections can be defined as being 'reversed', which means that their path has te be resolved backwards. For the regular virtual connections, we generate a `virtuallyConnected/4` term, and for the reversed connections, we generate a `reverseVirtuallyConnected/4` term. The first three arguments are similar to the concrete connections, the name of the relation, the source type, and the target type. However, the fourth argument is used to store the path to the target in the form of a list of tuples. For these lists and tuples, we use the built-in Prolog syntax. The list stores pairs of relation-type pairs. The relation part is used to navigate the PVA model, whereas the type is used to restrict the potential destinations to the types which lie on the defined path. By also storing the type as well, the ProbLog grounder is able to reduce the size of the program by a factor 2, which resulted in a large improvement of the analysis time.

For the inference of attack step success probability and defence availability likelihood, we have to refer to the transformed probabilistic code, which we will discuss in the next section. Each `AttackStepType` and each `DefenceType` instance causes the generation of `attackStepSucceeded/4` and `defenceIsFunctioning/1` rules. These rules verify the existence of an instance of the type which they model, and whether their evaluation should be executed. Additionally, the rules are used to implement the aforementioned 'hard' evidence. Finally, the transformation of the probabilistic property is invoked to obtain a reference to the rule denoting the evaluation of that piece of code in ProbLog. Using the name of this property, we are able to refer to the probabilistic derivation it entails, and by conjunction we add it to the `attackStepSucceeded` and `defenceIsFunctioning` rules.

On the template side, we have two types of connections for type instantiations. On one hand, we have internal connections, which associate type instantiations within a template. On the other hand, we have defined external connections, for associations between different templates. As the external connections are only relevant for the PVAI transformation, we do not generate ProbLog facts for those connections. For the `InstantiationConnection` class which represents internal connections, we generate `instantiationConnection/4` facts. These facts store the relation name, two tuples representing the kind and name of the instantiations, and finally the template to which the connection applies.

We have two non-trivial transformations from the PVAI model ProbLog. Specifically, we will describe the transformation of evidence, and the connections between templates. By the definition of the aforementioned 'hard' evidence structure, the transformation of the PVAI `Evidence` instances amounts to the generation of a single fact for each piece of evidence. From the references to the PVA model, the transformation script obtains the required names for the specification of the `instance/4` predicate used in the evidence specification.

Connections between template instances are based on the `ConnectionTemplate` class. The transformation consists of the generation of the concrete connections instances which will connect instances of concrete types. The role name is obtained from the `ConcreteConnection` definition as stored in the `ConnectionTemplate` instance. The fields required for the `instance/4` terms are all contained in the fields of the referenced source and target `TypeInstantiation` instances, and in the referenced `TemplateInstance` instances. Our current implementation of this transformation does not support virtual connections between instances within templates. This feature was not implemented due to time constraints, although its implementation can be achieved using a similar approach as the regular connections.

5.4.6 Probabilistic code implementation for ProbLog

Our transformation of the probabilistic code starts by invoking the transformation rule for the top-level element of the probabilistic expression. By using the declarative features from ETL, our parent probabilistic code classes specify whenever they require their children to be transformed. The resulting transformation sequence generated by the ETL engine corresponds with a post-order traversal of the probabilistic code tree.

In order to correctly perform the transformation of the probabilistic code objects from the PVA model to ProbLog, we have to make sure that our implementation corresponds with the original semantics from CySeMoL. To accomplish this, we have designed our PVA model (and the transformation from CySeMoL) in such a way that our probabilistic code is 'CySeMoL-agnostic'. We use the fact that P^{2}AMF derivations in CySeMoL are single expressions, with independent probabilistic components. This means that the order of evaluation of the probability distributions within a single expression is not relevant, which leaves us with the problem of encoding the first-order logic, and the probabilistic components of these expressions. We will now explain for each class from our PVA model how it is transformed to ProbLog.

We have to be careful to use the correct scope in our expressions. In CySeMoL, and in our PVA model, the expressions of different concrete classes have to be independent. To guarantee this, we assign a unique label to each generated ProbLog term, which ensures that ProbLog will consider those terms separately. This label is procured by increasing a counter, which simultaneously guarantees that all labels remain unique up to the bounds of the counter. Furthermore, all components of our expressions need to be aware of their local scope. The probabilistic code blocks are defined for a specific concrete types, however, we need to be able to identify multiple instances of those types. We solve this issue by passing the `instance` definition of the owning type instance as an argument to each generated ProbLog term of the probabilistic code. An example of such an instance definition is as follows:

```
instance({Kind},{InstantiationName},{InstantiationType},{TemplateName}).
instance(assetType,applicationServer,applicationServer,web_Server).
```

For probabilistic properties, the transformation script first checks whether an implementation is provided. Depending on the outcome, and the type of the owner of the property (either an attack step or a defence), the transformation script initializes a unique property name for reference by other parts of the ProbLog code. When an implementation is present, the implementation is first transformed, and its ProbLog rule is referenced from the property rule. This results in one of the following three lines of ProbLog code, where `e_ID/1` is the root expression instance of the probabilistic code implementation:

```
% With implementation
property_ID({INSTANCE}) :- e_ID({INSTANCE}).
% AttackStepType without implementation
attackStepSucceededProperty_ID({INSTANCE}).
% DefenceType without implementation
1/2::defenceProperty_ID({INSTANCE}).
```

We start with one of our leaf nodes, constants. In P^{2}CySeMoL, only the $\top$ and $\bot$ constants are used. However, P^{2}AMF allows the use of arbitrary floating point literals in the range $[0,1] \subset \mathbb{R}$. We use the variable probability annotation feature from ProbLog. For a constant class with attribute $c \in [0,1] \subset \mathbb{R}$, we generate the following ProbLog rule:

```
P::const_ID(Context) :- P is c.
```

For performance reasons, we only generate a single `const_ID` rule for each constant. This greatly reduces the size of the ProbLog program, and does not change its semantics, as constants independent by definition. Another optimization is the special treatment of probability values of 0.0 and 1.0. Instead of using variable probability annotations, we directly assign a value of $\top$ or $\bot$ using the ProbLog constants `true/0` and `false/0`:

```
const_ID(Context) :- true.
const_ID(Context) :- false.
```

The advantage of this notation is that the ProbLog grounder is able to quickly optimize programs containing `true` and `false` predicates. This is due to the propagation of the truth values to the other rules which depend on these rules. Note that the constants will never depend on their input, which is why the argument of the term is not used in the right-hand side of the rule.

For `And` nodes, we generate a new predicate, which is a conjunction of the predicates generated by the children of the and node. Given two child predicates `a(X)` and `b(X)`, we generate the following ProbLog code:

```
and_ID({INSTANCE}) :- a({INSTANCE}), b({INSTANCE}).
```

`Or` nodes are treated in a similar way as `And` nodes. However, we chose to refrain from using the or-operator (`;/2`), and define two rules with the same name instead. This is purely for aesthetic reasons, as the ProbLog semantics of both approaches are the same. Given two child predicates `a(X)` and `b(X)`, we generate the following ProbLog code:

```
or_ID({INSTANCE}) :- a({INSTANCE}).
or_ID({INSTANCE}) :- b({INSTANCE}).
```

We have implemented the negation of probabilistic expressions in our PVA model. This is also possible in ProbLog, however, we need to be careful for the negation-as-failure interpretation of ProbLog. As the analysis only reasons about elements within the PVA model, we are able to conform to the closed-world assumption of ProbLog. Therefore, we implement negation through the `not/1` function of ProbLog. For a child predicate `a(X)`, we generate the following ProbLog code:

```
not_ID({INSTANCE}) :- not(a({INSTANCE})).
```

In contrast to some popular Prolog engines, ProbLog does not have a dedicated if-then-else function. However, we can transform our `IfThenElse` nodes by generating multiple rules for each outcome of the condition. When the condition evaluates to true, we require that the 'then' branch is evaluated, and if the condition evaluates to false, we evaluate the 'else' branch. For a condition `c(X)`, a then-block `t(X)` and an else-block `e(X)`, we generate the following ProbLog code:

```
ite_ID({INSTANCE}) :- c({INSTANCE}), t({INSTANCE}).
or_ID({INSTANCE}) :- not(c({INSTANCE})), e({INSTANCE}).
```

With the goal of reducing the size of the transformation script, we have defined some static auxiliary rules which are added to the ProbLog program after the transformation. Two of these rules are used to reduce the code required to verify whether an attack step has succeeded, or whether a defence is operational. The transformation of the `AttackStepSucceeded`, `DefenceOperational` and `ConnectionAvailable` classes uses these helper rules to implement their semantics in ProbLog. The resulting rules which can be called from regular probabilistic code results have the following form, where `PATH` denotes the list of path navigation tuples as described for virtual connections:

```
% ConnectionAvailable
checkConnectionExists_ID({INSTANCE}) :-
  {INSTANCE}, checkConnectionExists({INSTANCE}, target, {PATH}).
% DefenceOperational
checkAttackStepSucceeded_ID({INSTANCE}) :-
  {INSTANCE}, checkAttackStepSucceeded({INSTANCE}, target, {PATH}).
% AttackStepSucceeded
checkDefenceWorking_ID({INSTANCE}) :-
  {INSTANCE}, checkDefenceWorking({INSTANCE}, target, {PATH}).
```

The final leaf nodes from the probabilistic code in the PVA model are the probabilistic distribution references. We will discuss the implementation of the probability functions in section 5.4.4, but first, we will focus on how these distributions are referenced within the ProbLog program. Probability distributions are unique up to their names within a probabilistic code block of the PVA model. This means that for each call of a probability distribution, we need to make sure that it is independent from other calls. This is achieved by generating predicates with a unique name, by including a identifier similar to the other rules. However, as the context of the code is no longer required for the independent evaluation of the probability distributions, the generated predicates no longer have any arguments. The generated predicate is specified in such a way that it can be proven with the probability equal to the probability distribution it references. We accomplish this using the auxiliary definitions of the probability functions. An example probability distribution reference has the following form:

```
dRef_ID({INSTANCE}) :- d_ID2.
d_ID2 :- logNormalCDF(4.85, 0.98).
```

Finally, we want to provide access to the evaluation of the probabilistic properties based on an instance of the owning the attack steps or defences. In other words, given an instance definition for an attack step or defence, how will we provide a means to evaluate the transformed probabilistic code for that instance? For defences, this is directly implemented using the `defenceIsFunctioning/1` rules. These rules can be referenced for any given instance. For the attack steps however, we require some additional processing to conform to the CySeMoL semantics. For each attack step, we generate a `attackStepSucceeded/4` rule, which references the probabilistic property. The arguments of this rule are the target asset name and type, the attack step type, and the instance in which the attack step is defined. The rule enforces the evaluation of the reachability of the attack step instance, meaning that it requires that the attack step is either an entry point or that an enabling attack step is reachable instead. This requirement is evaluated by means of the static `somePathToAttackStep/4` predicate, which conjuncts each generated rule.

5.4.7 Generating ProbLog programs from ProbLog models

The final step in the generation of the alternate analysis using ProbLog, is the transformation of the ProbLog model into a text program. We have developed a model-to-text transformation script using the Epsilon Generation Language (EGL), which transforms the generated ProbLog models into text. For implementation details on the transformation to text, we refer to the source code of the transformation script.

Our transformation process generates two separate ProbLog models, one for the PVA model, and another one for the PVAI model. We are able to transform both models to text, which result in partial ProbLog programs. These partial programs are then merged to create the complete program. This separation of transformation is reflected in the following modes of operation for the analysis generator:

1. Only transform a PVA model into a partial ProbLog program.
2. Only transform a PVAI model into a partial ProbLog program.
3. Construct the complete analysis ProbLog program.

Using this separation, it is possible to transform only the PVAI model to ProbLog, based on a previously transformed PVA model. Consequently, when the PVAI model changes, we do not need to transform both the PVA and the PVAI models, but are only required to transform the PVAI model to a ProbLog model, and transform that model to ProbLog.

We have implemented the distinction between transformations by generating multiple files, which are labelled '`X_pva`', '`X_pvai`' or '`X_complete`'. When we merge the results from both model-to-text transformations, we also include the handwritten ProbLog code, which is stored on the analysis generator classpath. This auxiliary code is comprised of the queries, the definitions of the probability distributions, and the ProbLog code, which couples the PVA with the PVAI model, by deriving PVA instances based on template definitions from the PVAI model. The resulting file is a valid ProbLog program and, when executed using the default mode in ProbLog, derives the marginal probabilities of all attack steps and defences.

5.4.8 Reporting analysis results

When the analysis completes, we would like to report the results to the user in a meaningful way. The current feedback to the user consists of the output provided by ProbLog. This output comes in the form of solutions to the queries, which relate to the components defined in the object model. The predicates of the queries are constructed in such a way that the result provides the following information for each attack step:

1. The attack step which succeeded.
2. The (named) target of the attack step.
3. The asset type of the target of the attack step.
4. The template in which the attack step target is defined.
5. The probability of success.

And for each defence:

1. The name of the defence instance.
2. The (named) target of the defence.
3. The template in which the defence is defined.
4. The probability of a success defence.

From this information it is possible to construct a graphical representation in a similar fashion as EAAT. Because of time limitations, we were unable to implement a graphical interface to the results of our analysis.

Another form of feedback is listing the most likely attack paths. At this moment, it is not possible to efficiently search for the *n* most probable paths using ProbLog, and therefore, searching for the most probable paths using ProbLog amounts to finding all possible paths with a positive probability. For ProbLog, this results in a state space explosion, where most effort is spent in the ProbLog grounder searching for all possible paths.

Chapter 6

Measurements and Evaluation

6.1 Introduction

In the previous chapters, we have introduced an approach to performing vulnerability analysis on CySeMoL models. We use a model-based approach, and have as a result arrived at using pivot model. The design of this model has been laid out in chapter 3. From this pivot model, we have reconstruct the vulnerability analysis of CySeMoL using ProbLog, as described in chapters 4 and 5.

For this research, we have set the three goals (see section 1.3 for details), stating that we aim to achieve the following:

1. An improved vulnerability analysis speed
2. Analysis input which can be automated
3. An analysis which is extensible

In order to verify whether our proposed approach to performing the vulnerability analysis of CySeMoL using ProbLog accomplishes our goals, we have performed experiments, which we will describe in this chapter. First we have determined the behaviour of the analysis configuration parameters of P^2CySeMoL, which we describe in section 6.2. Using the resulting configuration parameters, we compare the performance of P^2CySeMoL to our ProbLog implementation for different model sizes, which we will discuss in section 6.3. Finally, in section 6.5 we will provide our interpretation of the results from our measurements.

The specification of our experimentation environments is listed in table 6.1. Note that we use a slightly different environment for the ProbLog experiments, in comparison to the CySeMoL measurements. The reason for this, is that the SDD library used by ProbLog is not available for Windows. To circumvent this problem, we have decided to run the ProbLog experiments under GNU/Linux using a virtual machine running on the same host system as used for the CySeMoL experiments.

Table 6.1: The specifications of the machine on which the experiments where performed.

Part	*Host machine*	*Guest machine*
Operating System	Windows 10	Ubuntu 14.04.1 LTS
CPU	Intel Core i5-5200U @ 2.20GHz	Intel Core i5-5200U @ 2.20GHz
Logical cores	4 (hyperthreading)	2
Memory	8GB	2GB
Storage	200GB SSD	16GB SSD

Table 6.2: The observed statistics for the RMSE of models of different sizes.

Model	*Model size*	*Measurements*	*Mean RMSE*	*RMSE Standard deviation*
1	6	5	2.2	0.51
2	12	5	2.2	0.39

6.2 CySeMoL sampling parameters

6.2.1 Introduction

To ensure that we conduct a fair comparison between the analysis provided by CySeMoL and our own, we have conducted two experiments to determine suitable comparison parameters for the sampling algorithm of CySeMoL. Recall from section 2.3.6, that CySeMoL utilizes the Metropolis-Hastings algorithm. The two main configuration parameters of this algorithm are:

1. n, the total amount of samples drawn for the model.
2. b, the amount of burn-in samples.

First, we have determined whether the number of samples needs to be adjusted for the size of the model. In addition, we have established a number of samples which provides us with sufficient accuracy for inferring probabilities in CySeMoL models. We use the resulting values for n and b to run an execution time comparison between CySeMoL and our ProbLog implementation.

6.2.2 Scaling the amount of samples with respect to the model size

Our first experiment aims to discover whether CySeMoL requires more samples for larger models, or that n resembles the amount of times the *entire* model is sampled. For this experiment, we constructed two models. The first model contains a single attacker, directly connected to 6 web servers. The second model contains 12 web servers. Assuming that the total amount of samples indicates the amount of times the entire model is samples , when sampling with a low amount of total samples, we expect that increasing the amount of components will not lead to an increase in the error of the estimations. We measure the error using the root-mean-squared-error (RMSE):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(t_i - x_i)^2}$$

For our sampling parameters, we defined $n = 500$, and $b = 10$. Our null-hypothesis states that *the RMSE's are equal,* e.g. increasing the model size does not increase the error due to a lack of samples. By taking 5 samples of the RMSE of each model, we obtain the values as listed in table 6.2. Our smallest sample size is equal to 5, which corresponds (the RMSE samples follow a t-distribution) to a p-value of 2.132 for 95% confidence. This provides is with the following confidence interval for the difference in means:

$$(2.2 - 2.2) \pm 2.132\sqrt{\frac{0.51^2}{5} + \frac{0.39^2}{5}} \approx [-0.61, +0.61]$$

This interval includes the value of 0, which means our observations are not significant at the 95% confidence level, and we are unable to reject our null-hypothesis. Because of this result, we assume that we do not need to adjust the total amount of samples when the model size increases.

6.2.3 Appropriate sample size

For our experiments we need to account for the difference in accuracy between CySeMoL and ProbLog. Where our method using ProbLog infers the exact probabilities, CySeMoL is only able to provide approximations (which are in addition rounded to percentiles), where the accuracy is determined by the total amount of samples n.

In the previous section, we have concluded that we do not need to account for the size of the model when choosing a value for n. Therefore, we can suffice with the determination of a single value for n for all other experiments. Moreover, for practical purposes, security experts might only require a rough indication of risk, instead of the exact probabilities.

Starting with a sample count of 5000, we determined the accuracy of CySeMoL by calculating the RMSE for 10 measurements using this sample size. For this experiment, we created a model of a single attacker, who is connected to a single `ApplicationServer` instance.

After taking 10 measurements of the `ExecuteArbitaryCode` estimation, we obtained the following RMSE:

$$RMSE_{5000} \approx 0.008$$

which roughly reflects an error of a single percentile, which we deem acceptable for our comparison purposes. Consequently, our expectation is that a value of 5000 samples will provide sufficient accuracy for our comparison.

6.3 Analysis execution time measurements

6.3.1 Experiment design

Our main goal concerns improving the analysis speed of CySeMoL models. We use a scaling experiment in order to compare the performance of our ProbLog analysis approach, to the analysis performance of P^{2}CySeMoL.

Our scaling experiment is based on the minimal working example (see figure 2.6). We created copies of this model, where we gradually increase the amount of network components, and thereby the amount of attack steps and defences which have to be taken into account.

For our model size N, we use the total amount of `ApplicationServer` objects in the model. To increase the size by one, we connect two new `NetworkZone` objects to the existing `PhysicalZone` object, and connect a new `ApplicationServer` object to these network zones. The operating system and software product of the application servers is shared for all `ApplicationServer` objects. See figures 2.6 and 6.1 for the models of sizes $N = 1$ and $N = 3$.

For CySeMoL we measure the analysis time by the value reported by EAAT. For ProbLog, we use the Linux `time` command, which measures the difference in clock time between the start and end of its execution. We measure each model size at least 4 times, and determine the 95%-confidence interval of the execution time, based on those measurements.

6.3.2 Measurement results

Before selecting ProbLog as a potential target for replacing the CySeMoL analysis, we measured its execution time on hand-crafted models. From these experiments, we obtained the results shown in figure 6.2. In this graph, ProbLog is shown to perform faster than all other methods for our input models. These results have led to our decision to further examine the scalability of ProbLog when applying our analysis method to full-scale CySeMoL models.

We have run experiments with our new method up to $N = 20$. The results of our timing experiments are shown in figure 6.3. The difference with the old measurements is noticeable, and we will pay attention to these results in section 6.5.1. From these results, we observe that our method is faster for small values of N, but its performance worsens for larger values when compared to CySeMoL. From our measurements, CySeMoL appears to scale linearly for these model sizes. Our method on the other hand exhibits exponential scaling behaviour.

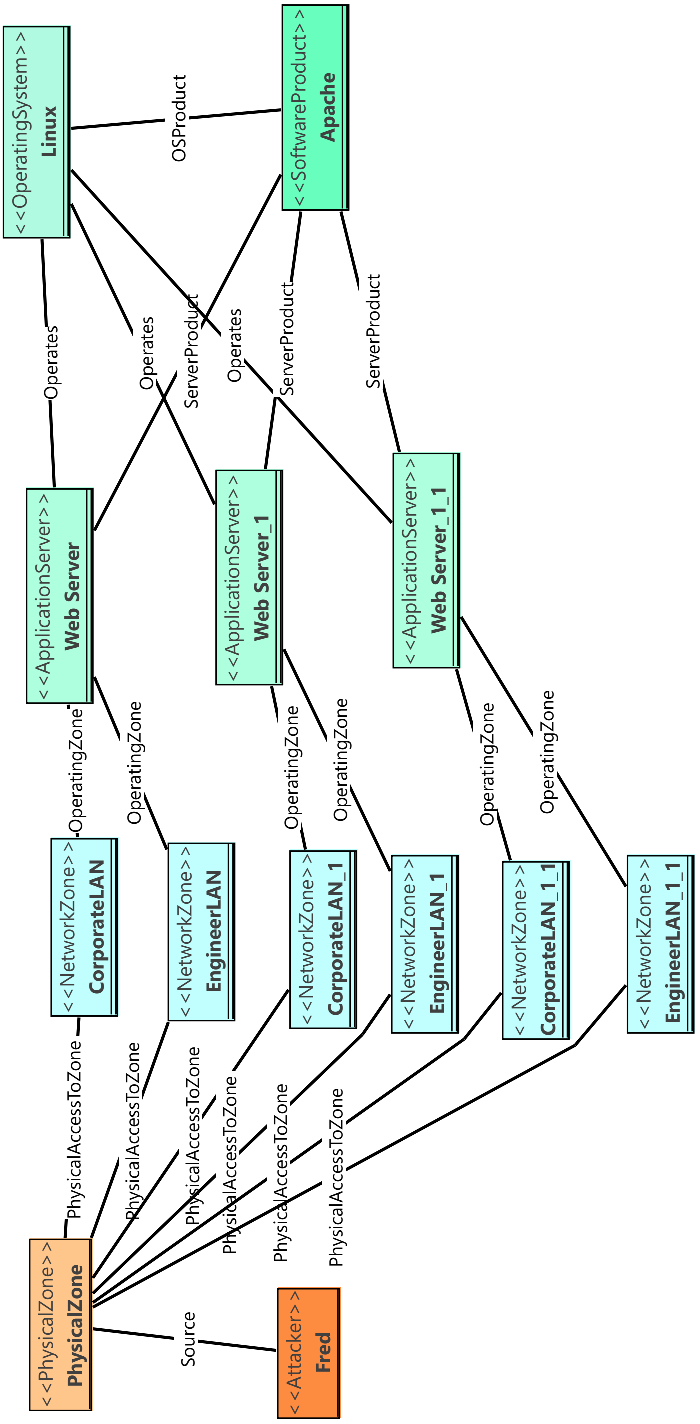


Figure 6.1: The model used for the analysis execution time measurement for $N = 3$.

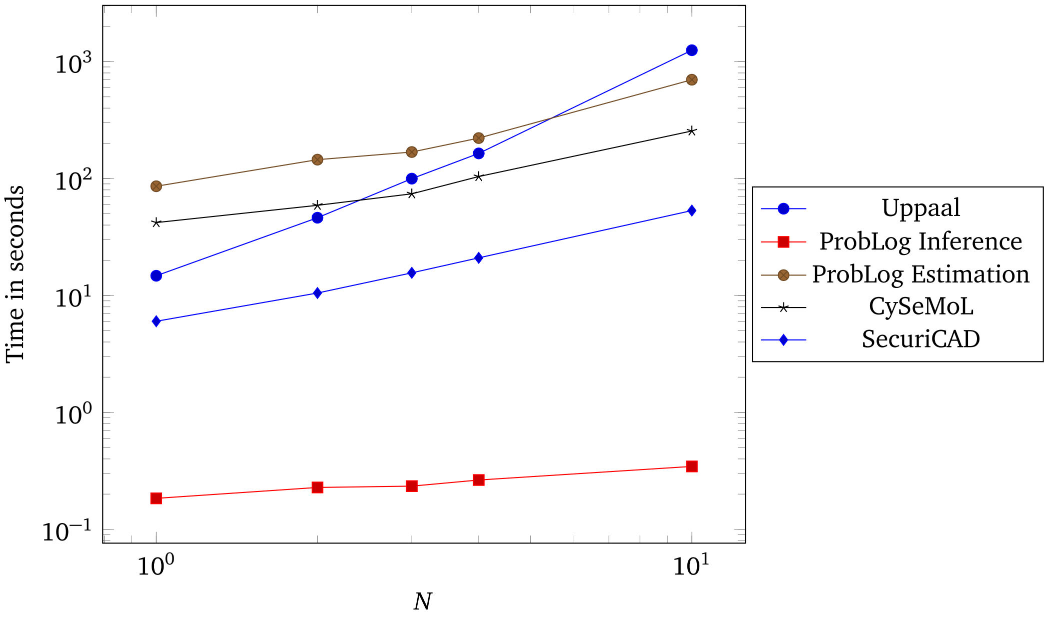


Figure 6.2: An overview of the execution times (on the logarithmic y-axis) of four analysis methods for $N = 1 \dots 10$ (on the logarithmic x-axis).

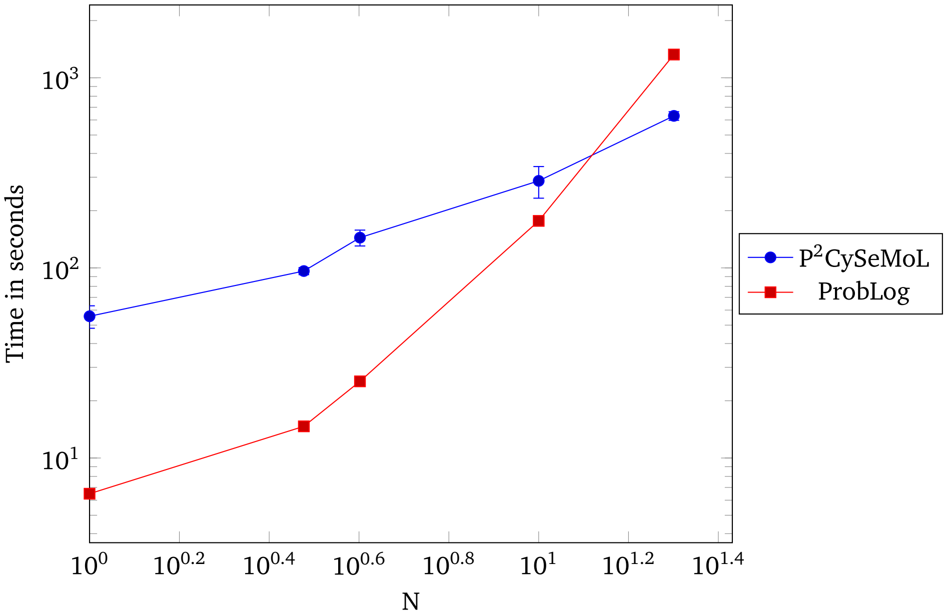


Figure 6.3: The measurement results of the scalability of CySeMoL versus ProbLog, along with their 95% confidence intervals. The x and y axes are logarithmic.

6.4 Analysis correctness

We have verified the correctness of our exact answers by comparing the results of ProbLog with the results of CySeMoL for a number of test models. By drawing 10000 samples, we ensure that the RMSE of CySeMoL becomes insignificant with respect to the significance of the results reported by CySeMoL (which amounts to two significant digits). The test models are the following:

1. The scaling experiment model of size $N = 1$.
2. The scaling experiment model of size $N = 4$.
3. The scaling experiment model of size $N = 1$, with an extra introduced (independent) attack step.
4. A model constructed by arbitrarily adding CySeMoL templates, and connecting them.

In these tests, the results of our ProbLog analysis match the results of CySeMoL, up to the accuracy supported by CySeMoL. From these results, we conclude that our ProbLog analysis is likely to be correct for valid CySeMoL models.

6.5 Discussion

6.5.1 Measurement results

Clearly, the performance of our new implementation differs significantly from our old implementation. We believe this is caused by the hand-crafted nature of the derivation rules. The definition of our new analysis shifts more work to the relatively slow ProbLog grounding phase, as more complex paths within the rule system have to be considered. For instance, the destinations of all relevant references (concrete and virtual connections) in the PVA model have to be resolved. This difference in performance does serve as inspiration for possible improvements, as we will mention in section 8.2.

From the resulting measurements of our method and the analysis using P^{2}CySeMoL, we conclude that our proposed approach using ProbLog has a worse scaling performance. We investigated the distribution of time for some of the ProbLog phases, the parsing, grounding, and the SDD analysis in particular. The appalling conclusion from this investigation is that ProbLog spends most of the analysis time in the grounding phase, rather than the probabilistic inference phase. This is interesting, given the fact that the grounding has the similar properties as a non-probabilistic reachability search using ProLog, which is known to be fast[14]. After enquiring the authors of ProbLog, we learned that the grounder has been re-implemented in the Python language, as part of the new SDD framework and the transition from YAP-Prolog[12]. As a consequence, the grounder of ProbLog “can definitely be a bottleneck"[16].

Another interesting result is that both ProbLog and CySeMoL fail to identify the potential for the reuse of calculations in the testing models. For instance, the scaling of our models results in some properties which all should have the same probability. A possible cause for this failure is our implementation of the P^{2}AMF to ProbLog transformation. From our experiments, we noticed ProbLog on one hand, is able to use only a single processing core, whereas CySeMoL is able to use all available processing power, which might result in a linear performance advantage. However, given that we measured the performance of CySeMoL and ProbLog using linear scaling, this would only result in a repositioning of the break-even point.

CySeMoL has a higher memory usage than our ProbLog implementation. For the largest model, P^{2}CySeMoL uses about 2 gigabytes of heap memory, where ProbLog uses a maximum of 700 megabytes for the same model. This might be an indication that we have not reached a performance bottleneck for CySeMoL where one might be reached for even larger models. The reason that we have not run experiments with larger models, or with a lower memory limit, is that the performance of both CySeMoL and ProbLog does not meet our requirements at those model sizes.

As a side note, CySeMoL does not report any accuracy statistics on its analysis. From experiments we learned that it is possible to determine a suitable number of samples for any level of accuracy by hand. Using our method, we are able to infer the exact probabilities of success for an attacker. A consequence of this, is that the probabilities inferred using our method are more reliable, as they will not vary between analyses. Therefore, we conclude that we have improved the accuracy of the vulnerability analysis of CySeMoL models.

6.5.2 Model robustness

Near the completion of the design of the PVA model and the transformation from the PRM to the PVA model, we discovered the existence the derived edges feature of EAAT. Our model proved its extensibility, as we were able to implement this feature in the PVA model by using inheritance, and were not required to redesign the entire model. We did however require an additional parser grammar in order to read the path specifications from the PRM. Moreover, we were able to add a new attack step to the PRM using EAAT, and this attack step was correctly identified and transformed to the PVA model by the iEaat Parser.

During the design of our models, we have encountered some problems with CySeMoL and EAAT as described in section 6.5.4. Our models and transformation tools have been designed to be able to cope with these bugs in EAAT and CySeMoL, and allow us to continue the analysis in the presence of these bugs. The user is notified of the presence of these bugs, and the applied solution. Note that these bugs might be fixed in the future, and that we are able to design around these bugs because we have their existence became known to us during our design process. However, the fact that we were able to find these bugs speaks positively for our development process.

6.5.3 Goal evaluation

We will now backtrack to the goals we set in section 1.3. Using the results described previously, we are able to reflect on the extent to which we have achieved our goals. We will examine our goals one by one, and assess if and to what extent we have achieved each individual goal.

Goal 1 - Improved vulnerability analysis speed We set out the goal to improve the analysis time of CySeMoL models. Our proposed analysis has a worse performance when compared to P^{2}CySeMoL for larger models. For smaller models however, our analysis is much faster. As we did not achieve the intended speed improvements for large models, we have failed to meet this goal.

Goal 2 - Automatable analysis input We have succeeded in achieving this goal by designing and implementing the PVA and PVAI models, which capture the data and semantics of CySeMoL models. The `IEAATParser` program enables us to convert any CySeMoL model to a combination of PVA and PVAI models. Therefore, a specific network infrastructure input can be constructed either by transforming an existing CySeMoL model, or by creating an instance of the PVAI model. By using a combination of EMF and XMI, the automated generation of input models is straightforward. For instance, a network scanner could generate an XMI file containing a PVAI model, which is then transformed to ProbLog using our *analysis generator* tool.

Goal 3 - Extensible analysis In order to make our models accessible, adaptable, and extensible, we have implemented our models using EMF (specifically XMI and Ecore). This technology facilitates the use and accessibility our models. On the transformation level, it is possible to completely replace our probabilistic analysis with a different analysis. Our approach separates the transformation to ProbLog in two stages with a pivot model inbetween. The first stage generates PVA and PVAI models, and the second stage generates ProbLog models, which allows us to investigate different analysis methods, or to extend our ProbLog method. Furthermore, the robustness of our approach was proven during design, as we were able to discover and cope with model extensions, and bugs in the EAAT implementation (see the additional discussion in section 6.5.2).

For testing and comparison purposes, we have used the same attack time parameters as CySeMoL, including the workdays parameter. The adaptability of our ProbLog transformation code facilitates the design of a completely different interpretation of the probability distributions. As we only store the information contained in the original P^2AMF code, it is possible to construct alternate analyses using the same probability distributions. As an example, we are now able to replace the analysis using workdays with a calculation using the expected means of the probability distributions.

6.5.4 Encountered problems with CySeMoL and EAAT

During our work with CySeMoL, we have encountered some bugs and deficiencies. The authors at KTH have been notified of these issues, which means that these issues might be resolved in the future. Here we provide an overview of the most notable ones. Our developed tools either detect or work around these issues, which ensures their correct operation as far as possible.

CySeMoL model errors The CySeMoL model contains two classes, `BypassDetectionSystems` and `NetworkVulnerabilityScanner`, who have not been assigned a parent class. We have manually assigned a parent class to them, based on their role in the Class Modeller. From these observations, we determined that `BypassDetectionSystems` is supposed to be an instance of the `AttackStep` class, and `NetworkVulnerabilityScanner` should be an instance of the `Defense` class.

EOM (de)serialization errors During the implementation of a parser for the EOM model, we tested the semantics of the serialization format of attack steps. We noticed that too little information is stored to reconstruct the correct attack step. This leads to errors when the file is deserialized. The object modeller is unable to determine the correct attack step, and assigns an arbitrary attack step instead. This can be observed when assigning the `ServerConnectTo` attack step where, after saving and loading, the attack step type is changed to `ServerAccess`, which also leads to different outcomes of the vulnerability analysis.

Another problem with the serialization of entry points, is when an attacker is assigned multiple entry points to a single target. In this case, one of the entry points is not serialized, which leads to an incorrect analysis when the model is reloaded.

Finally, when a template contains multiple instantiations of the same type (for instance, two instances of the `PortSecurity` class), evidence of their properties is serialized based on their class. This leads to situations where it is possible to store evidence that a property is both true and false, which is obviously not what a designer usually intends. Furthermore, when such a model is loaded, only the last evidence is used, which means that some evidence of the model can be lost.

Template attributes When a template has some attributes defined, it is possible to delete the object which owns the referred attributes. However, when this object is removed, the reference in the template is not cleaned up, which results in a dangling reference. This bug is not inherent to the CySeMoL model, but is a mistake in the Class Modeller. The bug prevents the analysis from taking place without first repairing the dangling reference.

Chapter 7

Related Work

7.1 Attack Graph analysis

The vulnerability analysis of CySeMoL bears some similarities with the work on threat modelling using attack graphs and trees[50]. This similarity is likely caused by earlier work of the authors on *Bayesian Defense Graphs*[75], which resemble attack trees much more closely. On an abstract level, CySeMoL generates a probabilistic attack graph for the currently defined network architecture, and searches for potential paths. Most other attack tree formalisms reason from the perspective of the goals instead of the attacker[50], on the other hand, the automated reasoning of CySeMoL is performed from the perspective of an attacker, and from there, determines which goals that attacker is able to reach.

Hermanns et al. propose a method for the analysis of attack-defence diagrams using Stochastic Timed Automata[34]. This work extends on earlier methods for performing risk analysis using priced timed automata, incorporating both time and cost[51]. The vulnerability analysis of CySeMoL incorporating both the time required for attacks, as well as the probabilistic components can also be modelled with Stochastic Timed Automata. Where the analysis of Kumar et al. uses Uppaal CORA[3] for PTAs, the approach in [34] uses the Modest tool set[32] for STAs. Hermanns et al. pay more attention to the game-theoretic approach of threat modelling, by incorporating costs (e.g. monetary) into their models, and by searching for attack paths optimal in both time and cost.

7.2 Probabilistic Programming

ProbLog has been developed as part of research in the domain of probabilistic programming. On an abstract level, the analysis of CySeMoL using P^{2}AMF can be interpreted as a probabilistic program, including the support for observations by means of evidence. Apart from ProbLog, there exist more tools from the probabilistic programming domain, which might be suitable for the efficient implementation of the CySeMoL analysis.

In preparation for the development of an alternative analysis engine, we assessed whether we could perform the vulnerability analysis of CySeMoL using Uppaal SMC[6]. An extension of the Uppaal model checker[4], which supports stochastic model checking of timed models. Uppaal has been used for the analysis of attack trees[51], and the SMC extension would allow the definition of probabilistic models. For the inference of probabilities, Uppaal SMC currently resorts to sampling methods, which suggests that its performance would be similar to CySeMoL. Nevertheless, the support for timed models allows for a more elaborate modelling of the time required for performing attacks on a network.

The PRISM probabilistic model checker[53] is a model checker with support for nondeterministic, probabilistic, and timed models, as well as the model checking of Discrete and Continuous Time Markov Chains (DTMCs and CTMCs). Similar to Uppaal, PRISM provides analysis facilities for Probabilistic Timed Automata (PTA) models. PRISM supports two types of analysis, one based on *abstraction refinement*, and the other on *symbolic* methods using Binary Decision Diagrams. Using these methods, it is possible to obtain PRISM defines a language for the description of the models it supports[63]. The language is used to specify modules, in which states are described through the use local and global variables and clocks. Transitions are described using so-called 'guarded commands', which specify one or more probabilistic or nondeterministic transitions which are only enabled when a given expression (the guard) holds.

PITA (Probabilistic Inference with Tabling and Answer subsumption)[68] is a method for inferring probabilities in probabilistic logic programs. PITA has been implemented as an extension to XSB[78], an implementation of Prolog which leverages tabling (the memoization of proven facts) to improve its performance. The language provided by PITA is able to express Logic Programs with Annotated Disjunctions[82]. Similar to ProbLog, PITA supports estimating the probabilities of predicates using a specialized predicate `prob(atom,P)`, which enumerates the probability of `atom` in `P`.

MulVAL[62] is an open source vulnerability analysis tool. The tool uses Datalog[8], a subset of the Prolog language. Using Datalog, the authors define a logic program, which is used to describe the *network infrastructure*, *exploits*, *attack steps*, and *security policies*. Due to the design of ProLog, MulVAL serves as a system of facts and inference rules, which enables the tool to determine security policy violations. MulVAL is able to generate traces of attack steps which lead to those violations. MulVAL collaborates with OVAL[87] compliant vulnerability scanners to obtain vulnerability information on hosts within the network under analysis. This makes MulVAL suitable for finding vulnerabilities in real-world networks. MulVAL is built upon XSB[71], a Datalog implementation which provides tabled resolution and supports multi-threading.

7.3 Model transformation for analysis

Model transformation provides a generic approach to translate between different models. This flexibility allows the definition of multiple analyses by transforming models to representations of formal reasoning systems, for which the analyses can be defined. For instance, Wolters[88] defines a model transformation to transform attack trees to Timed Automata[4]. Next, Uppaal is used to perform a timed reachability analysis for components of the original attack tree.

A similar approach is used by Anastasakis et al.[2]. Their research focuses on the transformation of UML models with OCL constraints to the textual modelling language Alloy[40]. The Alloy framework provides the Alloy Analyser, a tool which is able to automatically discover design flaws in software. Anastasakis et al. employ their model transformation to be able to apply the analysis by the Alloy Analyser on arbitrary UML models.

Chapter 8

Conclusions and Recommendations

8.1 Conclusions

We have been able to successfully reconstruct the vulnerability analysis of CySeMoL using a different analysis method. In this paper, we have presented a model-based approach which provides accessible, robust and extensible models for representing CySeMoL models, and its vulnerability analysis. We have succeeded in extracting this information from arbitrary serialized CySeMoL models.

By means of model transformations, we have defined a method for reconstructing the CySeMoL vulnerability analysis using probabilistic logic. Our approach allows us to infer the exact likelihood an attacker to succeed in compromising components of the input network. Furthermore, we were successful in maintaining the separation between the network component definitions, and the network architecture definitions. Due to our usage of metamodelling, model transformations and model-to-text transformations, we have kept the definition of the vulnerability analysis separate from the definition of the network infrastructure. As a result, when the network architecture changes, only those changes need to be evaluated by our tools.

We have compared the asymptotic behaviour of the time required to perform a vulnerability analysis using our ProbLog method to the analysis provided by CySeMoL. Although our approach using ProbLog is faster for smaller models, it exhibits worse performance for larger models when compared to CySeMoL.

Summarizing, we have achieved two out of our three goals, as shown in table 8.1. Due to our successes in goals 2 and 3, TNO is investigating how our model-based approach and solution to their problem can be integrated with the results from an automated topology scan.

Table 8.1: An summary of the status of our goals.

No.	*Goal description*	*Succeeded*
Goal 1	Improved vulnerability analysis	×
Goal 2	Automatable analysis input	✓
Goal 3	Extensible analysis	✓

8.2 Recommendations and Future Work

We have succeeded in achieving most of our goals. Given our successful development of an extensible approach to the development of a vulnerability analysis, we are able to investigate alternative options for the analysis using ProbLog. Due to time constraints, we have focused on reproducing the analysis provided by CySeMoL. A limited amount of attention has been dedicated to the improvement of the realism, extent or soundness of the analysis. In this section, we will discuss some of the research paths which are yet to be explored, and other potential improvements to our work.

Most of the analysis time of our ProbLog implementation of the vulnerability analysis of CySeMoL models is spent in the grounding phase (see section 2.7.6). From discussions with the authors of ProbLog[16], we learned that the grounder has been re-implemented, and can very likely be improved. This claim is supported by the fact that during the implementation, most performance gains were achieved by indirectly simplifying the grounding problem. In addition, the authors of ProbLog have suggested[16] the possibility for removing the grounding phase by directly generating logic formulae representing the analysis from the PVA and PVAI models.

The current conversion of P^2AMF to ProbLog ensures total independence between derivations. Even though this is desirable, it has the side-effect of eliminating the sharing of calculations between derivations, even in the cases where this does not change the outcome. We would like to investigate whether we can alter our transformation such that the sharing of results between derivations is optimized, while retaining the independence of probabilities. We have already implemented one such an optimization by reusing the generated predicates for the 'true' and 'false' expressions from P^2AMF.

Our method calculates the exact probabilities of success for an attacker. This level of accuracy might not be required for the intended purposes. Therefore, it might be worthwhile to investigate whether (preferably bounded) approximate methods can improve the analysis speed. During the research topics phase, we have already investigated whether Uppaal SMC might be suitable for this task, however, it was unable to outperform ProbLog at that time.

The problem of determining the probability distribution of success based on time investment is still an open problem. We have introduced some flexibility for different mechanisms of determining success probabilities, yet we are still required to resort to single probabilities instead of probability distributions. CySeMoL contains the information for the inference of such distributions, as we have observed during our experiments with Uppaal SMC. However, analysis of such (hybrid) probabilistic models is still subject to ongoing research[21, 59, 86, 32].

CySeMoL couples the notion of the time investment by the attacker with his chance for success. The explicit usage of time might make timed models better suited for probabilistic analysis. However, we do not constrain the passage of time in any way, which might indicate that probabilistic timed model checking is not required. It remains that we need to take into account varying time investments for different kinds of attack steps, as mentioned before.

Aside from ProbLog, we have also investigated PITA as a candidate for our analysis engine. PITA is an extension to Datalog, which also supports the evaluation of Logic Programs with Annotated Disjunctions. From [67, 66, 68], we concluded that ProbLog had the potential to be better suitable, as it seemed to perform better on our intended class of problems (large probabilistic graphs) with respect to PITA. With our current results, we cannot dismiss the possibility that the current implementation of ProbLog might not be suitable for our performance goals. As PITA supports the same type of annotated programs as ProbLog, we believe we should examine the performance of our analysis when we substitute ProbLog with PITA as our inference engine.

The transformation from CySeMoL to PVAI takes more time than necessary. A lot this time can be attributed to the serialisation and deserialisation of the PVA model. This process might be improved in a number of ways. For instance, using compression will reduce the time required for I/O operations. Another method would be to replace the XMI serialization with a more efficient binary format, although this would reduce the portability of the generated PVA and PVAI models.

Bibliography

[1] Christopher J Alberts and Audrey Dorofee. *Managing information security risks: the OCTAVE approach*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.

[3] Gerd Behrmann and KG Larsen. Uppaal-cora, 2005.

[4] Glenn Behrmann, Alexandre David, Kim G Larsen, John Hakansson, Paul Petterson, Wang Yi, and Monique Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[5] Gunnar Björkman. The viking project–towards more secure scada systems. 2011.

[6] Peter Bulychev, Alexandre David, Kim Gulstrand Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. Uppaal-SMC: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*, 2012.

[7] Markus Buschle, Pontus Johnson, and Khurram Shahzad. The enterprise architecture analysis tool–support for the predictive, probabilistic architecture modeling framework. In *19th Americas Conference on Information Systems, AMCIS 2013; Chicago, IL; United States; 15 August 2013 through 17 August 2013*, pages 3350–3364. Association for Information Systems, 2013.

[8] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):146–166, 1989.

[9] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[10] Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[11] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[12] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[13] Adnan Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332. Citeseer, 2004.

[14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

[15] Anton Dries. Problog documentation. `http://problog.readthedocs.io/en/latest/`, August 2016. Accessed on 14 september 2016.

[16] Anton Dries. Problog mailing list. `https://ls.kuleuven.be/cgi-bin/wa?A0=PROBLOG`, October 2016. Accessed on 13 october 2016.

[17] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 312–315. Springer, 2015.

[18] Sven Efftinge, Peter Friese, Arno Hase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, et al. Xpand documentation. Technical report, Technical report, 2004-2010.(cited on page 64), 2004.

[19] Mathias Ekstedt. Foreseeti website. `https://www.foreseeti.com/`, 2016. Accessed on 11 november 2016.

[20] Mathias Ekstedt, Pontus Johnson, Robert Lagerstrom, Dan Gorton, Joakim Nydren, and Khurram Shahzad. Securi CAD by foreseeti: A CAD tool for enterprise cyber security management. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*, pages 152–155. IEEE, 2015.

[21] Christian Ellen, Sebastian Gerwinn, and Martin Fränzle. Statistical model checking for stochastic hybrid systems involving nondeterminism over continuous domains. *International Journal on Software Tools for Technology Transfer*, 17(4):485–504, 2015.

[22] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz–open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[23] Thomas Erl. *Service-oriented architecture: a field guide to integrating XML and web services*. Prentice Hall PTR, 2004.

[24] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[25] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(03):358–401, 2015.

[26] Elizabeth Spangler Flint. The cobol jigsaw puzzle: Fitting object-oriented and legacy applications together. *IBM Systems Journal*, 36(1):49–65, 1997.

[27] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[28] Rune Fredriksen, Monica Kristiansen, Bjørn Axel Gran, Ketil Stølen, Tom Arthur Opperud, and Theo Dimitrakos. The CORAS framework for a model-based risk management process. In *Computer Safety, Reliability and Security*, pages 94–105. Springer, 2002.

[29] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[30] Carla P Gomes, Joerg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI*, pages 2293–2299, 2007.

[31] Object Management Group. XML metadata interchange specification. `http://www.omg.org/spec/XMI/`, June 2015. Accessed on 7 september 2016.

[32] Arnd Hartmanns and Holger Hermanns. The modest toolset: an integrated environment for quantitative modelling and verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 593–598. Springer, 2014.

[33] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[34] Holger Hermanns, Julia Krämer, Jan Krčál, and Mariëlle Stoelinga. The value of attack-defence diagrams. In *International Conference on Principles of Security and Trust*, pages 163–185. Springer, 2016.

[35] H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. P^{2}CySeMoL: Predictive, probabilistic cyber security modeling language. *IEEE Transactions on Dependable and Secure Computing*, 12(6):626–639, Nov 2015.

[36] Hannes Holm, Mathias Ekstedt, Teodor Sommestad, and Matus Korman. A manual for the cyber security modeling language. *Royal Institute of Technology (KTH), Tech. Rep*, 2013.

[37] Hannes Holm, Teodor Sommestad, and Mathias Ekstedt. CySeMoL: A tool for cyber security analysis of enterprises. In *22nd International Conference on Electricity Distribution (CIRED)*, 2013.

[38] Michael Howard and David LeBlanc. *Writing secure code*. Pearson Education, 2003.

[39] ISO ISO and IEC Std. Iso 15408-1: 2009. *Information technology-Security techniques-Evaluation criteria for IT security-Part*, 1, 2009.

[40] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[41] Tomi Janhunen. Representing normal programs with clauses. In *ECAI*, volume 16, page 358, 2004.

[42] Jetbrains. Kotlin programming language. `https://kotlinlang.org/`, September 2016. Accessed on 20 september 2016.

[43] Pontus Johnson, Johan Ullberg, Markus Buschle, Ulrik Franke, and Khurram Shahzad. An architecture modeling framework for probabilistic prediction. *Information Systems and e-Business Management*, 12(4):595–622, 2014.

[44] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[45] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer, 2005.

[46] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2014.

[47] Dimitrios Kolovos, Louis Rose, Richard Paige, and A Garcıa-Domınguez. The epsilon book. *Structure*, 178:1–10, 2010.

[48] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062, page 200, 2006.

[49] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[50] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review*, 13:1–38, 2014.

[51] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. Quantitative attack tree analysis via priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 156–171. Springer, 2015.

[52] Ivan Kurtev. State of the art of qvt: A model transformation language standard. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2007.

[53] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[54] Allan J Macleod. Algorithm as 245: A robust and reliable algorithm for the logarithm of the gamma function. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 38(2):397–402, 1989.

[55] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[56] Nenad Medvidovic, David S Rosenblum, David F Redmiles, and Jason E Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, 2002.

[57] Haralambos Mouratidis and Paolo Giorgini. Secure tropos: a security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(02):285–309, 2007.

[58] Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. Dsharp: fast d-dnnf compilation with sharpsat. In *Advances in Artificial Intelligence*, pages 356–361. Springer, 2012.

[59] Davide Nitti. Distributional clauses (beta) extension to problog2. `https://github.com/davidenitti/DC`, March 2016. Accessed on 29 march 2016.

[60] Object Management Group (OMG). Meta object facility (MOF) v2.5 specification. `http://www.omg.org/spec/MOF/2.5/`, June 2015. Accessed on 19 september 2016.

[61] Oracle. Simple api for xml. `https://docs.oracle.com/javase/tutorial/jaxp/sax/index.html`, August 2016. Accessed on: 15 august 2016.

[62] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *USENIX security*, 2005.

[63] Dave Parker. Prism manual. `http://www.prismmodelchecker.org/manual/`, February 2016. Accessed on 10 march 2016.

[64] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*, volume 49, pages 579–598. ACM, 2014.

[65] Bruce Potter. Microsoft sdl threat modelling tool. *Network Security*, 2009(1):15–18, 2009.

[66] Fabrizio Riguzzi. Speeding up inference for probabilistic logic programs. *The Computer Journal*, page bxt096, 2013.

[67] Fabrizio Riguzzi and Terrance Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[68] Fabrizio Riguzzi and Terrance Swift. The PITA system for logical-probabilistic inference. *Latest Advances in Inductive Logic Programming*, page 79, 2014.

[69] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2008.

[70] Jaideep Roy and Anupama Ramanujan. Xml schema language: taking xml to the next level. *IT professional*, 3(2):37–40, 2001.

[71] Konstantinos Sagonas, Terrance Swift, and David S Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Record*, volume 23, pages 442–453. ACM, 1994.

[72] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[73] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19, 2003.

[74] Alex Sellink, Harry Sneed, and Chris Verhoef. Restructuring of cobol/cics legacy systems. In *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, pages 72–82. IEEE, 1999.

[75] Teodor Sommestad, Mathias Ekstedt, and Pontus Johnson. Cyber security risks assessment with bayesian defense graphs and architectural models. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.

[76] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[77] Venkat Subramaniam. Programming groovy. *Dallas: The Pragmatic Programmers LLC*, 2008.

[78] Terrance Swift and David S Warren. Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[79] Sten-Åke Tärnlund. Horn clause computability. *BIT Numerical Mathematics*, 17(2):215–226, 1977.

[80] Nico M Temme. Computational aspects of incomplete gamma functions with large complex parameters. In *Approximation and Computation: A Festschrift in Honor of Walter Gautschi*, pages 551–562. Springer, 1994.

[81] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[82] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *Logic Programming*, pages 431–445. Springer, 2004.

[83] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *Workshop on Probabilistic Logic Programming (PLP), Vienna*, 2014.

[84] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with tp-compilation. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[85] John Von Neumann. 13. various techniques used in connection with random digits. 1951.

[86] Qinsi Wang, Paolo Zuliani, Soonho Kong, Sicun Gao, and Edmund M Clarke. Sreach: a bounded model checker for stochastic hybrid systems. *arXiv preprint arXiv:1404.7206*, 2014.

[87] Matthew Wojcik, Tiffany Bergeron, Todd Wittbold, and Robert Roberge. Introduction to OVAL: A new language to determine the presence of software vulnerabilities, 2003.

[88] NH Wolters. Analysis of attack trees with timed automata (transforming formalisms through metamodeling), 2016.

[89] Reinder Wolthuis and Frank Fransen. Segrid project website. `http://www.segrid.eu/`, February 2016. Accessed on 16 february 2016.

Appendix A

Metamodels

A.1 Overview

This appendix lists the diagrams of the PVA, PVAI and ProbLog metamodels. These metamodels define the structure of these models. In chapter 3, we elaborately discuss the design of each model.

A.2 The PVA metamodel

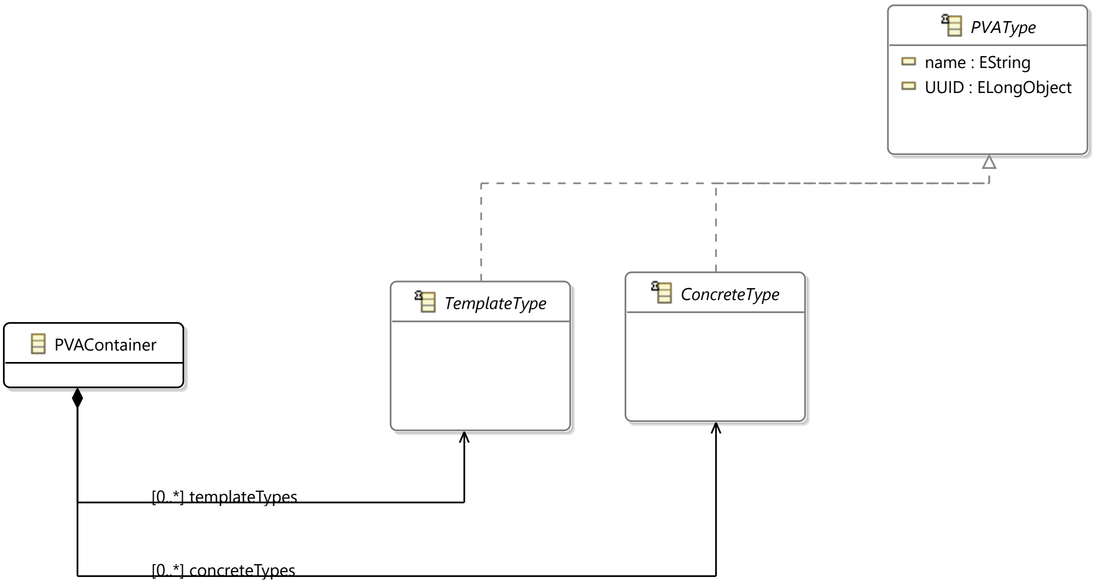


Figure A.1: The diagram of the root of the PVA metamodel.

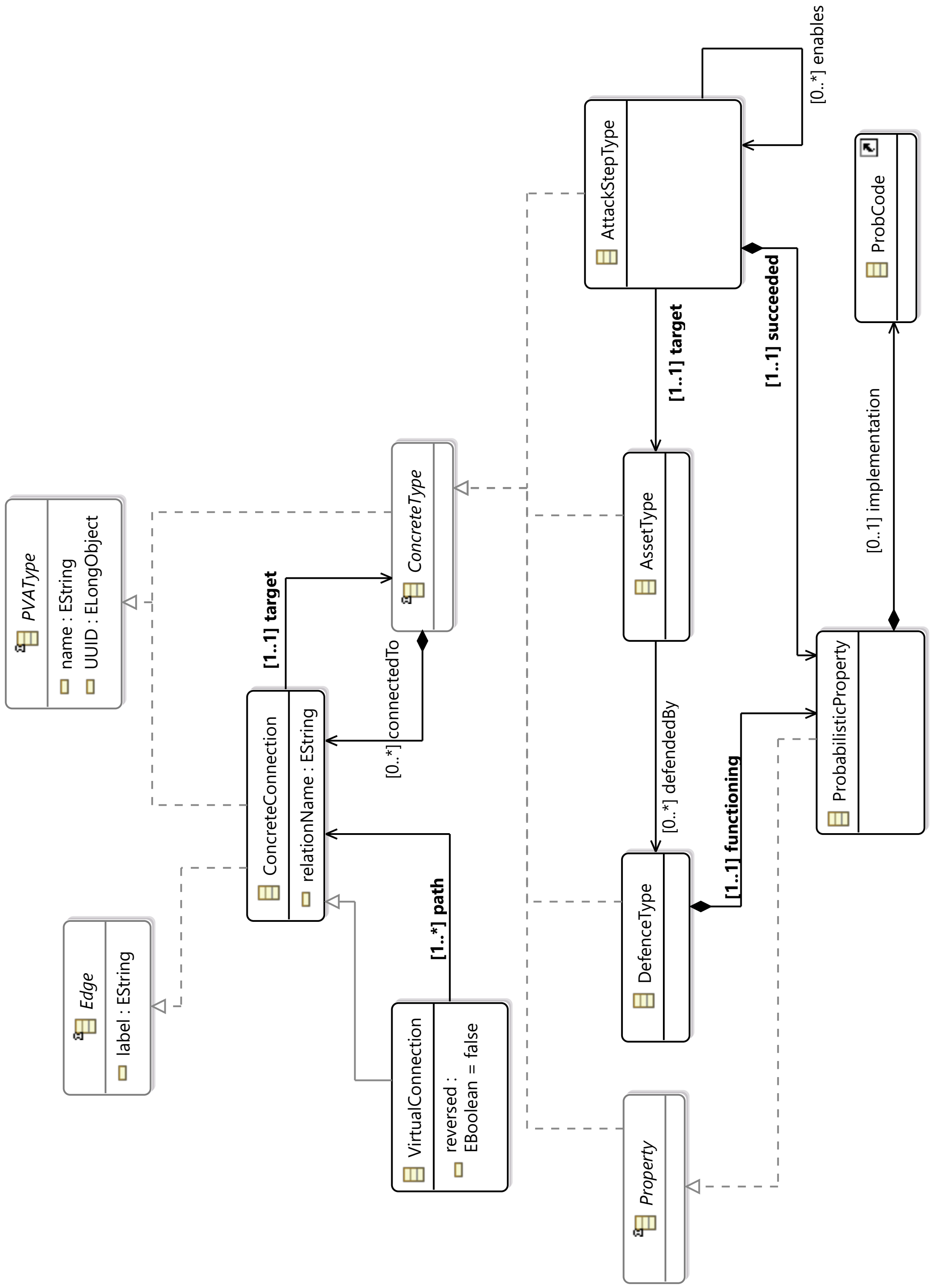


Figure A.2: The part of the PVA metamodel used for representing all concrete model objects.

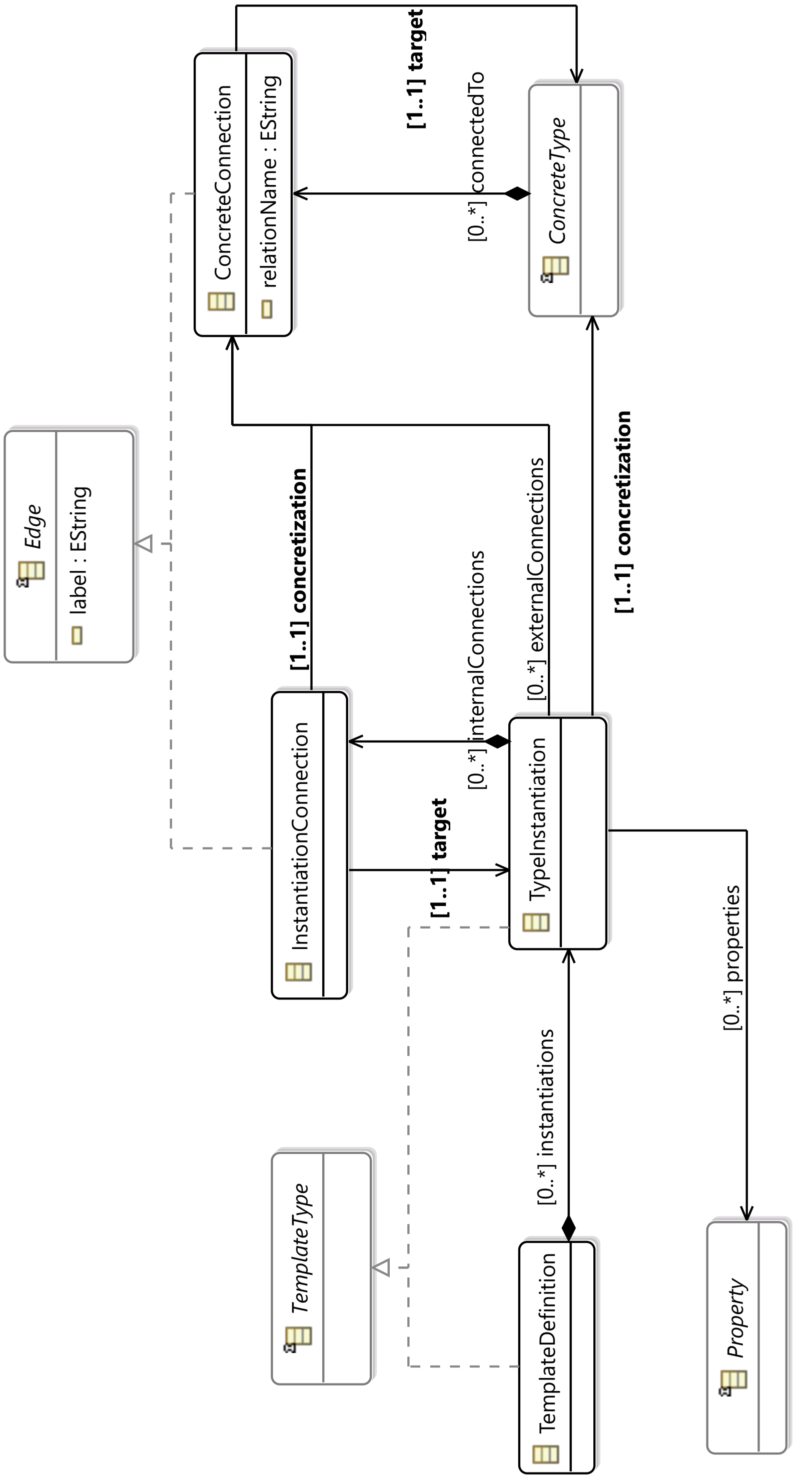


Figure A.3: The part of the PVA metamodel used for representing all template model objects.

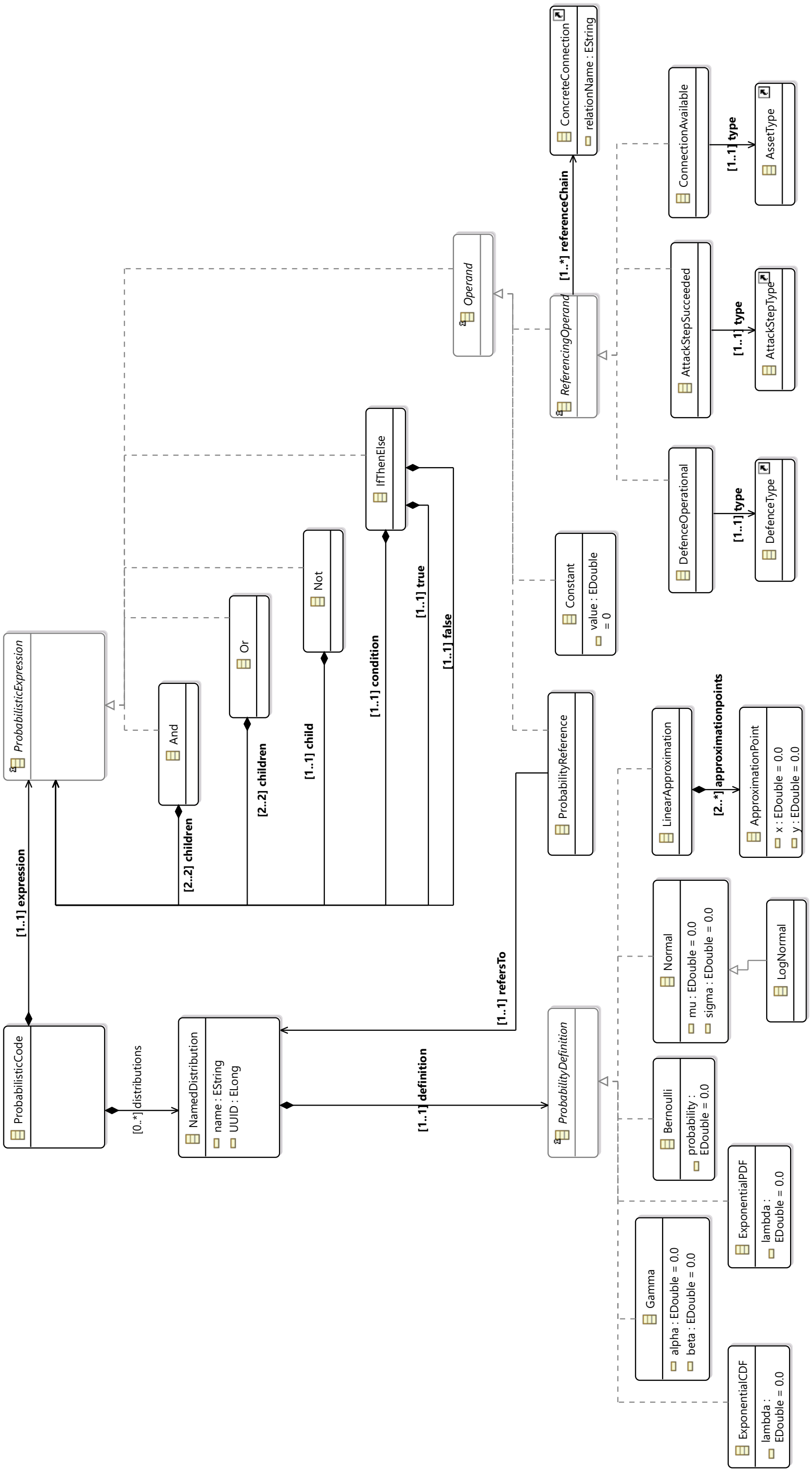


Figure A.4: The part of the PVA metamodel used for representing probabilistic P^{2}AMF programs.

A.3 The PVAI metamodel

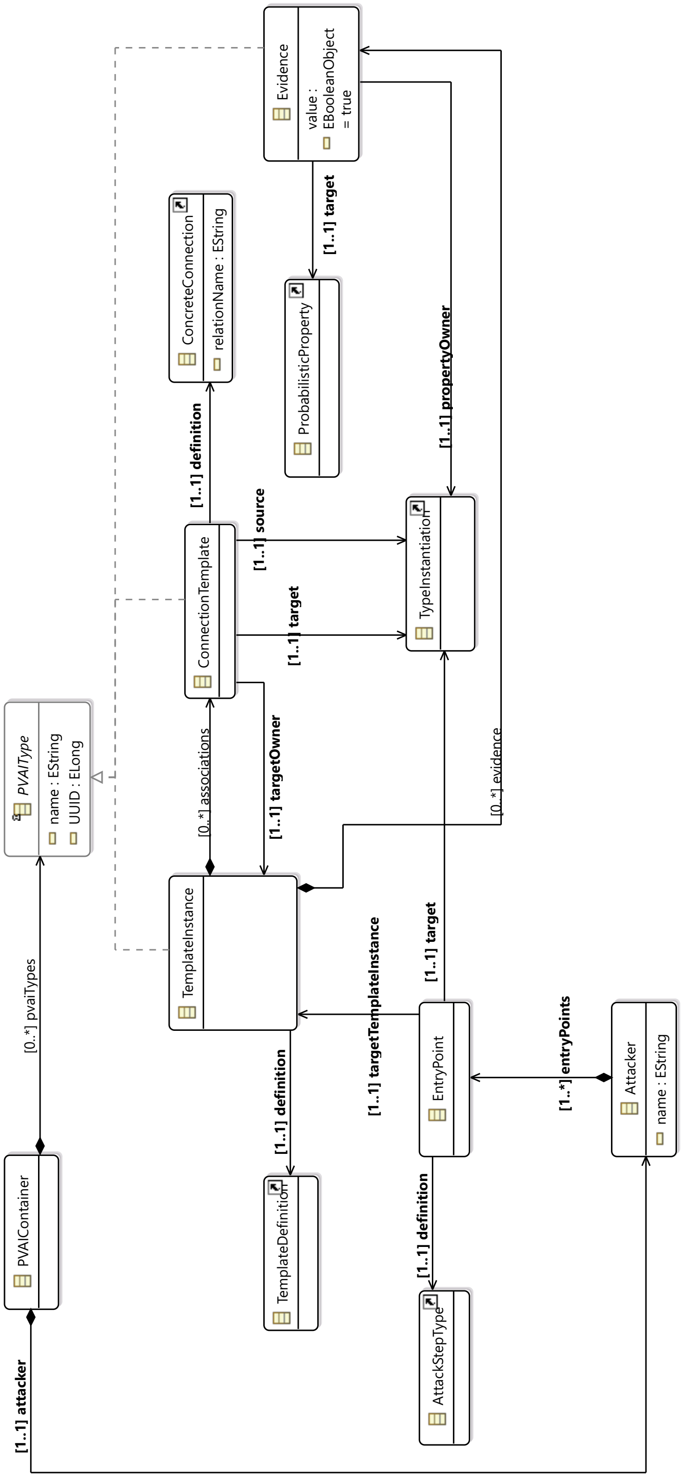


Figure A.5: The diagram of the Probabilistic Vulnerability Analysis Instance metamodel. Unfortunately the graph is non-planar ($K_{3,3}$ is subgraph of the diagram), which hampers its readability.

A.4 The ProbLog metamodel

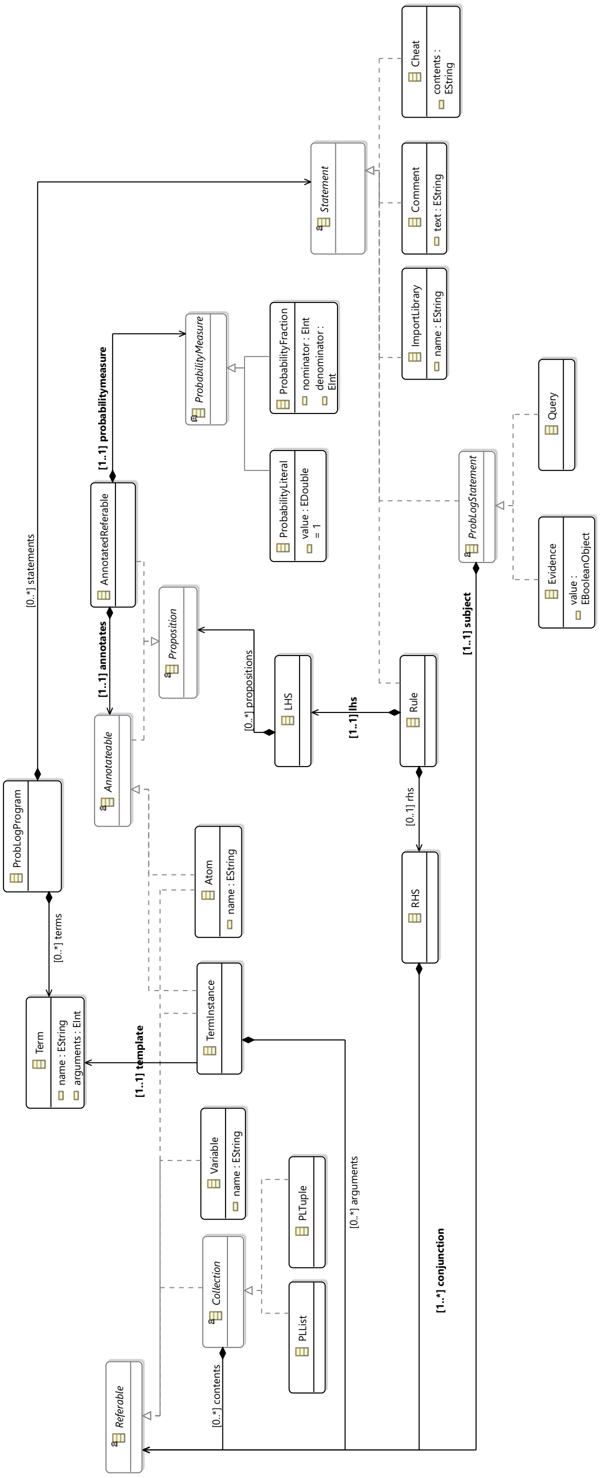


Figure A.6: The diagram of the metamodel used for modelling ProbLog programs.

Appendix B

Parser grammars

B.1 Overview

In this appendix, we list the actual ANTLR4 definitions of the grammars used to generate the parsers used by the IEAAT Parser program. The design and use of the parsers defined by these grammars is discussed in chapters 3 and 4.

B.2 General P^{2}AMF parser grammar

```
grammar P2AMF;

program :
    letblock* expression EOF
;

letblock:
    'let' ID ':' type '=' probabilitydistribution 'in'?
;

//Explicitly left-recursive, let ANTLR4 take care of it
expression:
     left=expression AND right=expression
    | left=expression OR right=expression
    | NOT expr=expression
    | ifthenelse
    | operands
    | '(' expression ')'
;

operands:  booleanoperand
              | numberoperand
              | reference;

ifthenelse:
    'if' condition=expression 'then'
        t=expression
    'else'
        f=expression 'endif';

```

```
probabilitydistribution:
    //Inline distributions wrapped in a bernoulli distribution
    'bernoulli' '(' probabilitydistribution ')'
    | ('self' '.')? (
        lineardist
        | expdist
        | lognormaldist
        | normaldist
        | gammadist
        | bernoullidist
    )
;

reference:
    attackstepEnabled
    | attackstep
    | defence
    | connectionavailable
    | ID;

connectionavailable: objectreference '->' 'notEmpty' '(' ')';

attackstep:
    'visited'
        '->' 'intersection' '(' objectreference ')'
        '->' 'notEmpty' '(' ')';

attackstepEnabled:
    'visited'
        '->' 'intersection' '(' 'self' '.' 'source' ')'
        '->' 'notEmpty' '(' ')';

defence: 'defenseAvailable' '(' objectreference  '->' 'asSet' '(' ')' ')';

objectreference:
    ('self' '.')? (ID '.')* ID;


//distributions
bernoullidist:
    'bernoulli'
        '(' p=numberoperand ')';

expdist:
    'exp'
        '(' lambda=numberoperand ',' attackertime ')';

lineardist:
    'linear'
        '(' x=arrayexpression ',' y=arrayexpression ',' attackertime ')';

normaldist:
    'normal'
        '(' mu=numberoperand ',' sigma=numberoperand ',' attackertime ')';
```

```

lognormaldist:
    'lognormal'
        '(' mu=numberoperand ',' sigma=numberoperand ',' attackertime ')';

gammadist:
    'gamma'
        '(' alpha=numberoperand ',' beta=numberoperand ',' attackertime ')';

arrayexpression:
    '[' numberoperand (',' numberoperand)* ']' ;

//constants
booleanoperand:
     TRUE
    |FALSE;

numberoperand:
    DASH? (intop | doubleop);

intop:
    NUMBER ;

doubleop:
    NUMBER '.' NUMBER;

type:
    'Real'
    | 'Bool'
    | 'Int';

attackertime:
    'Attacker' '.' 'Time';

//Named tokens
//Only define those which are required to be referenced from the AST
DASH: '-';
AND : 'and';
OR : 'or';
NOT : 'not';
TRUE : 'true';
FALSE : 'false';
CHAR: 'a'..'z' | 'A'..'Z';
NUMBER: DIGIT+;
DIGIT: [0-9];
ID : CHAR (CHAR | NUMBER | '_')* ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT : DASH DASH ~[\r\n]* -> skip;
```

B.3 Path parser grammar

```
grammar GetPaths;

paths :
    (
        attackStepList
        |
        '(' attackStepList ')'
        |
        TARGET putInSet
    ) DASH 'visited' EOF
;

attackStepList:
    attackStepReference union*;

attackStepReference:
    reference castToAttackStep putInSet;

reference:
    'self'? '.' ID;

union:    '->' 'union' '(' attackStepReference ')';

castToAttackStep:
    '.' 'oclAsType' '(' 'AttackStep' ')';

putInSet:
    '->' 'asSet' '(' ')';

//Named tokens
//Only define those which are required to be referenced from the AST
TARGET: 'target';
DASH: '-';
AND : 'and';
OR : 'or';
NOT : 'not';
TRUE : 'true';
FALSE : 'false';
CHAR: 'a'..'z' | 'A'..'Z';
NUMBER: DIGIT+;
DIGIT: [0-9];
ID : CHAR (CHAR | NUMBER | '_')* ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT : DASH DASH ~[\r\n]* -> skip;
```

B.4 Derived edge parser grammar

```
grammar VirtualEdges;

path:
    'self' ('.' ID)+ '->' 'asSet' '(' ')';

DASH: '-';
CHAR: 'a'..'z' | 'A'..'Z';
NUMBER: DIGIT+;
DIGIT: [0-9];
ID : CHAR (CHAR | NUMBER | '_')* ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT : DASH DASH ~[\r\n]* -> skip;
```

Appendix C

Command-line tools

C.1 Overview

We have developed two command-line tools which perform the transformation steps described in this thesis. In this appendix, we will show how the tools are used. The source code of these programs is hosted at the following URL:

```
https://github.com/SwiftPengu/ProbabilisticVulnerabilityAnalysis
```

C.2 iEaat Parser

The iEaat Parser has been packaged as a runnable JAR file. It can be run using the Java runtime environment with the following command-line instruction:

```
java -jar IEAATParser-1.10.jar <arguments>
```

When running the program without arguments, it outputs its usage information:

```
usage: IEAATParser <source> [-f] [-pva] [-pvai]
    -f      Force overwriting existing files
    -pva    Generate a PVA model file
    -pvai   Generate a PVAI model file
```

The `f` switch will overwrite any pre-existing `pva` or `pvai` files. If this option is left-out the tool will detect any existing PVA model, and generate a `pvai` file based on that model. The generation of models is specified through the `pva` and `pvai` switches.

For example, to generate both the PVA and PVAI models for a file named `Input.iEaat`, we can use the following command:

```
java -jar IEAATParser-1.10.jar Input.iEaat -pva -pvai
```

C.3 Analysis Generator

The analysis generator executes the transformation of pva and pvai models to the ProbLog model, and the model-to-text transformation to ProbLog code. Similar to the IEAATParser, the analysis generator is packaged as a runnable JAR file, and can be run by executing the following command-line instruction:

```
java -jar AnalysisGenerator-1.1.jar <arguments>
```

When running the program without arguments, it outputs the following usage information:

```
usage: AnalysisGenerator [-complete] -o <outputFile> [-pva <pvaSource>]
                         [-pvai <pvaiSource>]
    -complete                Whether a complete ProbLog file should
                             be generated.
    -o,--out <outputFile>    The name of the output file.
    -pva <pvaSource>         Whether to generate the PVA skeleton.
    -pvai <pvaiSource>       Whether to generate the PVAI rules and
                             queries.
```

The pva and pvai command-line arguments signal the transformation of the provided PVA or PVAI models to ProbLog. The base name of the output file is specified through the o/out parameter. When the complete switch is used, the generated ProbLog code for the PVA and PVAI models is merged into a single file, along with the auxiliary ProbLog code. The resulting file is a legal ProbLog program.

For example, to construct a complete analysis program from the generated files from the iEaat Parser example, we can use the following command:

```
java  -jar AnalysisGenerator-1.1.jar -o output -pva Input.pva
      -pvai Input.pvai -complete
```

This will generate a complete ProbLog program for the given PVA and PVAI models.

To re-use an existing result of a PVA model transformation, we use the following command:

```
java -jar AnalysisGenerator-1.1.jar -o output -pvai Input.pvai -complete
```

This will look for an existing file named output_pva.pl and generate a output_pvai.pl file. The output is merged with the auxiliary program to generate a complete ProbLog program.

The resulting complete program can be run by invoking ProbLog, which will instruct ProbLog to perform the probabilistic vulnerability analysis. For this, we use the following command:

```
problog output_complete.pl
```