

UNIVERSITÀ DEGLI STUDI DI GENOVA

Facoltà di Ingegneria



Corso di Laurea Magistrale in Ingegneria Informatica

A Framework for Designing Attack Strategies in Cyber Range Scenarios

Supervisor:	Prof. Gabriel Costa
Candidate:	Hossein Ghodrati
Student number:	3422289

July 2017

List of Figures	4
Abstract	6
1- Introduction	7
2 - Background	15
2.1 - Domain-Specific Languages	15
2.1.1 - Xtext	15
2.1.3 - Xtend	19
2.2 - Virtual Scenario Description Language[18]	19
2.2.1 - Scenario in VSDL	19
2.3 - Prism Model Checker	21
2.3.1 - Markov Chains	23
2.3.2.1 - Discrete Time Markov Chain (DTMC)	23
2.3.3 - Prism Modelling Language	24
2.3.3.3 - Prism Property Specification	27
3 - Design	29
3.1 - Overall Architecture	29
3.2 - Vulnerability specification in VSDL	30
3.3 - Code Generation	32
3.4 - Vulnerability Specification Scripting Language	33
3.4.1 - Head	34
3.4.2 - Body	34
3.4.3 - Reward	36
4 - Implementation	37
4.1 - Writing a Code Generator With Xtend	37
4.1.1 - Resource	39
4.1.2 - File System Access	39
4.1.3 - Implementation of compileModel method	40
4.1.3.1 - Xtend Method Declaration	41
4.1.3.2 - Template Expressions	41
5 - Use Case	43

5.1 - Injection Attack	43
5.1.1 - Scenario	43
6 - Conclusion and Future work	47
6.1 - Future work	47
Bibliography	48
Appendix A - VDSL syntax	50

List of Figures

Figure 1-1 Ignite Conference 2016: Conquering the Cyber Range	8
Figure 1-2 Cyber Range Team	10
Figure 1-3 Cyber Range Components	10
Figure 1-4 Example of Scenario	12
Figure 2-1 Xtext Entity Example	16
Figure 2-2 model Grammar	16
Figure 2-3 typedef Grammar	17
Figure 2-4 mapsto Grammar	17
Figure 2-5 extended Grammar	17
Figure 2-6 Cross Reference	18
Figure 2-7 total Grammar	18
Figure 2-8 VSDL Example	21
Figure 2-9 Model Checking	22
Figure 2-10 Probabilistic Model Checking	22
Figure 2-11 Prism Example 1	25
Figure 2-12 Prism Example 2	26
Figure 2-13 State Reward	27
Figure 2-14 Transition Reward	28
Figure 3-1 Abstract Architecture	30
Figure 3-2 Logical Flow	31
Figure 3-3 sql Injection	32
Figure 3-4 Specification Scripting Language	33
Figure 3-5 Specification Scripting Language -2	35
Figure 3-6 Specification Scripting Language -3	35

Figure 3-7 Reward	36
Figure 4-1 Eclipse Code Generator-1	38
Figure 4-2 VsdIGenerator	38
Figure 4-3 Node Filter	39
Figure 4-4 Network Filter	40
Figure 4-5 Code Generator-5	40
Figure 4-6 fsa	40
Figure 4-7 Method Declaration	41
Figure 4-8 compileModel method	42
Figure 5-1 Injection	44
Figure 5-2 Scenario	45
Figure 5-3 Prism Code	46
Figure 5-4 Total Reward Property-1	48
Figure 5-5 Total Reward Property-2	47
Figure 5-6 Result Total Reward Property	48

Abstract

Cyber Security is becoming more and more important in our world. Exposure to cyber threats is getting wider and more transverse. Developing new capabilities and new tools to improve Cyber Defense is a major challenge.

The most important issue in Cyber Defense is staff training, Cyber Ranges are used for this end. A Cyber Range is a virtual environment used for training and exercising. We have implemented a framework that allows us to convert the Cyber Range scenarios into Prism models, then we can analyze the Cyber range and optimize it.

1- Introduction

In today's society, Computer Security has become increasingly important in our lives. Exposure to cyber threats is wide and transverse. Its effects can be devastating for personal life, the whole country, and the whole world. As a consequence, developing new capabilities and new tools to improve **Cyber Defense** is a major challenge.

NATO Cooperative Cyber Defense Center defines **Cyber Defense** as a “proactive measure for detecting or obtaining information as to a cyber intrusion, cyber attack, or impending cyber operation or for determining the origin of an operation that involves launching a preemptive, preventive, or cyber counter-operation against the source”.¹

Clearly these activities strongly depend on the skills of the security experts. Thus, a **cornerstone** of Cyber Defense is **staff training**, which is done through laboratory activities and exercises, the design and evaluation of new tools. However, these activities must be realistic. In the other words, staff should not feel they are under training in a fictional environment: they must behave as in a real security event. **Cyber Ranges** are used for this purpose.

The Clusit (Italian Association for Computer Security) report², declares that 2016 was the worst year ever with regard to the evolution of cyber threats and their impacts.

In fact, 1,050 attacks, classified as “serious”, occurred during this year.

According to the study, the largest percentage of serious attacks in 2016 occurred in the healthcare sector (+ 102%), large distribution (+ 70%) and banks (+ 64%), with a significant impact on the victims in terms of economic damage, reputation and leakage of sensitive data.

Cyber Range

A Cyber Range is a virtual environment used for training and exercising in cyber security related areas. In this environment, staff is trained in using both defensive and attacking tools, tactics, and strategies. A Cyber Range can also be used for developing cyber technology. As we have already mentioned, a Cyber Range may be a virtual environment, but there are some alternatives. For instance, an Hybrid Cyber Range is a combination of actual and virtual components. This typology is especially useful for Cyber Physical System training, such as embedded or industrial systems.

¹ NATO Cooperative Cyber Defence center of Excellence: <https://ccdcoc.org/cyber-definitions.html>

² https://clusit.it/wp-content/uploads/download/Rapporto_Clusit%202016.pdf



Figure 1-1 Ignite Conference 2016: Conquering the Cyber Range

Cyber Range users can be divided into four groups:

- **Students.** To apply their theoretical knowledge in a simulated network environment, improve cyber skills, work as a team for solving cyber problems and preparing for Cyber Security Certifications
- **Educators.** Educators can use Cyber Ranges as a classroom for evaluating their students
- **Professionals.** They can be from different groups such as information technology, law enforcement, cyber security, incident handlers, that use Cyber Ranges for improving individual and team knowledge and skills.
- **Organizations.** They can use Cyber Ranges for evaluating their own proficiency, training their team and test new methods.

The exercise in a Cyber Range involves several **teams** with distinct and in some cases conflicting roles (Figure 1-2):

- A **red team** plays the role of attacker (malicious users), the red team will have to try to violate the security of this infrastructure by accessing a specific data or compromising a specific resource.
- A **green team** is responsible for the physical and online infrastructure.
- A **blue team** has the task of defending; the purpose, for example, could be to verify, and improve, the security of an infrastructure in a limited amount of time.
- A **Yellow Team** to improve realism, during the action, makes legitimate interactions with the environment; can be partially simulated by automatic tools.

Team **Tools** are classified in:

- **Red tools**: attack tools, such as scripts for exploitation, malware or backdoor to inject targets, products for interception of data flows, abnormal traffic generators and so on;
- **Blue tools**: tools for performing security analysis, management of incidents and digital forensics such as, for example, analyzers of vulnerabilities, monitoring tools, sandboxes and so on;
- **Yellow tools**: tools to manage security, improve defense perimeter or internal such as, for example, Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS), firewalls, antivirus, antimalware, security systems, etc.
- **Green Tools**: tools for Infrastructure Monitoring like Hypervisors, Routers, and so on.

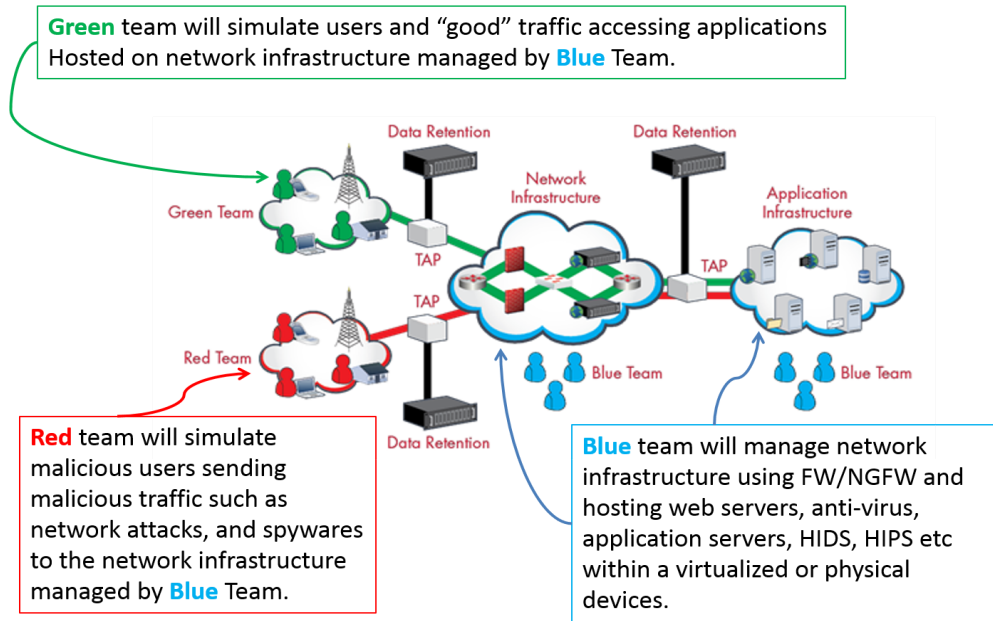


Figure 1-2 Cyber Range Team

A logical representation of the components needed to create a Cyber Range can be seen in Figure 1-3.

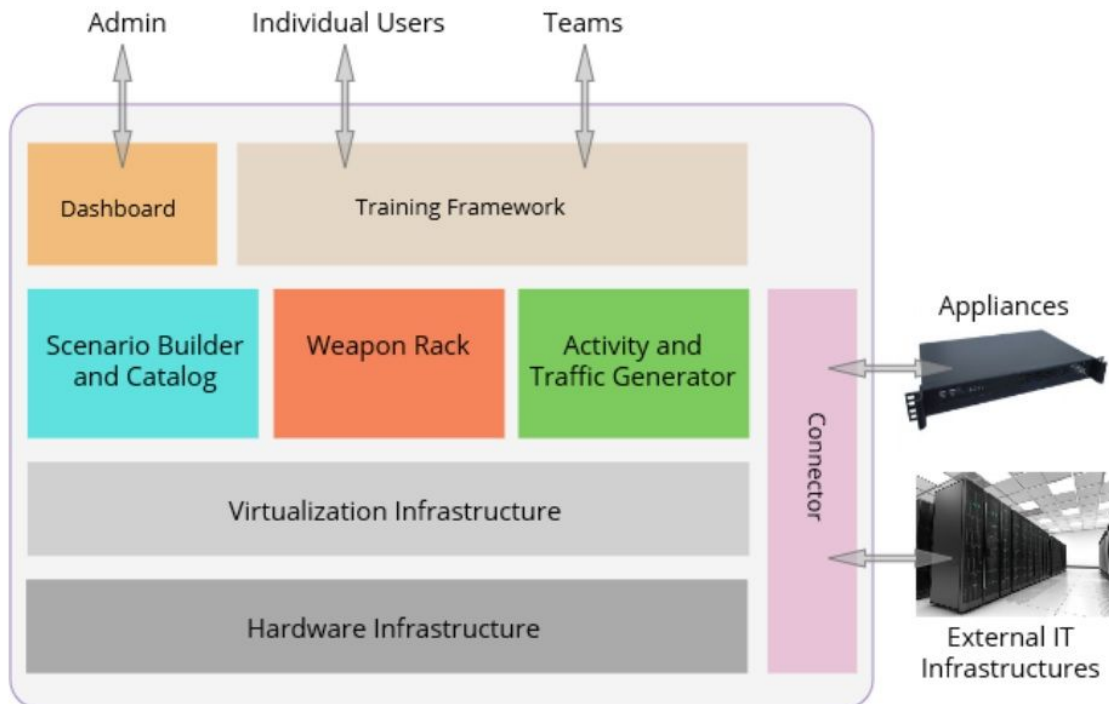


Figure 1-3 Cyber Range Components

- The **Training Framework** represents the user access point to the Cyber Range and realizes the capabilities to support training and training activities related to both individual training and team training.
- The **Dashboard** is the interactive tool, available to the administrator, for the management and monitoring of the Cyber Range.
- The **Weapon Rack** provides Cyber Range users with a set of state-of-the-art attack and defense tools to be used both in training and in real testing of equipment and systems.
- The module called **Activity and Traffic Generator** simulates, in automatic mode, the standard traffic of the infrastructure (see yellow team).
- The **Connector** allows the traditional architecture of the Cyber Range to extend towards a different schema, called hybrid Cyber Range architecture, for the purpose to insert into the simulated scenario hardware items or physical appliances.
- The **Virtualization** and **Hardware** Infrastructure are responsible for the software and hardware layer which implement and guarantee adequate performance.

The operating environment that includes networks, hardware, software and their behavior during practice sessions is called a **Scenario**. The scenario is the central element of every training session. As a matter of fact, the main purpose of the Cyber Range is to stage scenarios that meet the training requirements.

Consider, for example, the scenario shown in Figure 1-4 :

A network consists of four subnets called **Server Room**, **Laboratory**, **Meeting Room** and **Control Room**. Each subnet connects computational nodes (e.g., servers and laptops). Moreover attributes label some nodes of the scenario. In the **Server Room Network**, there are three nodes one characterized by a specific operating system, while the others contain a vulnerable service and a malware. In the **Laboratory Network**, two laptops are connected, one of which is accessible via remote shell, and a cell phone that can move into the Meeting Room network. The **Meeting Room network** contains a host with file server functions (i.e. SMB service). The **Control Room Network** contains a personal computer with active antivirus software and another PC that contains the data to defend (the “flag”).

The **blue** team performs its defense task within this perimeter for a certain period of time. Then it stops and switches to offline status.

The **red** team, on the other hand, acts after the blue team. Also, they cannot access the full network, but only the public subnet.

Red team's goal is to steal the data (i.e., to "capture the flag") and/or penetrate into the internal network, exploit pivoting and lateral movement techniques to reach the target node.

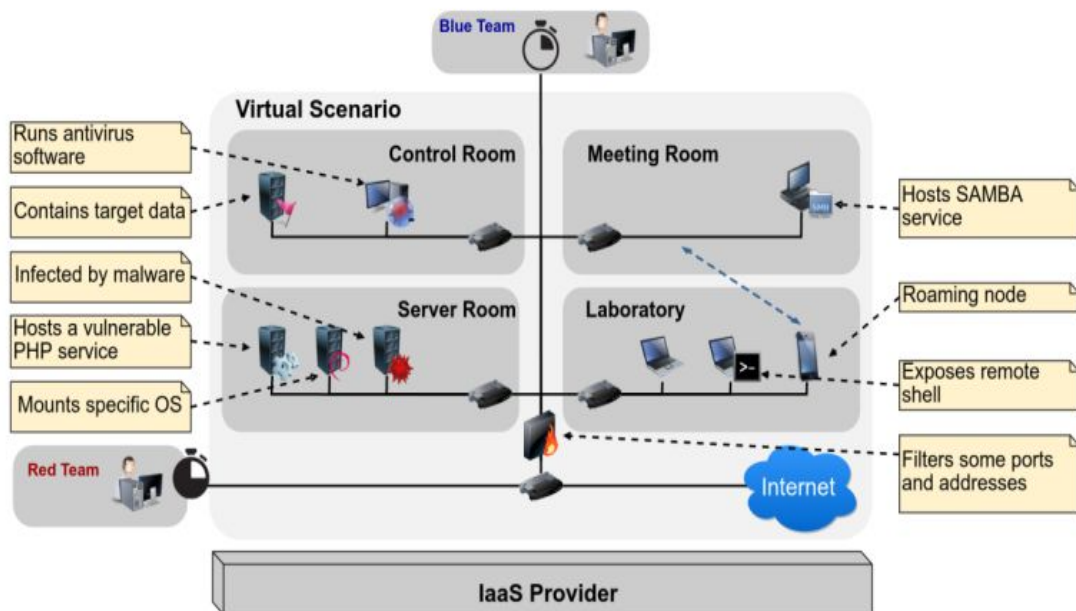


Figure 1-4 Example of Scenario

The relevant information to describe a scenario may belong to the following categories.

- **Networking:**
 - What are the existing channels and connections?
 - Are there active firewalls? What rules do they apply?
- **Hardware:**
 - What are the hardware capabilities of the nodes? (CPU, speed, disk size)
 - What is their role in the scenario? (e.g servers, mobile devices, phones, laptops, etc.)

- **Software:**
 - Which OS runs on a node? Which applications and libraries are installed?
- **Data:**
 - What information is stored in a node? Does the file systems contain any relevant file?
- **Users and privileges:**
 - Who can access a certain node? What are the privileges of the users over the file system of a node?
- **Time:**
 - How does the infrastructure change over time?

The **scenario builder** (As can be seen in Figure 1-3) is the element that deals with providing tools, or more precisely a framework for creating the scenarios with all features described above within the Cyber Range.

As we have already said, Cyber threats are increasing so fast, we've seen that Cyber Ranges have a key role in cyber Defence, To have **qualified Cyber Ranges** First of all, we must comprehend **attack strategies** well, only in this way we can design effective and well-done exercises in Cyber Ranges.

In this thesis, we have implemented a **framework** that allows us to **refine Cyber Ranges** scenarios. In detail, to write the scenarios, we used VSDL that we represented it in section 2.2, **we have extended the VSDL** synthesis, to have a new attribute of nodes that allows us to specify vulnerability type that we see in section 3.2, once we have written a scenario in VSDL, the framework gets information from the scenario and associated JSON file to type of vulnerability specified in the scenario, to have the consistent JSON files, we **created a script specification** that we see in section 3.4 and at the end, the core of our framework is **Code Generator** that creates the prism model of the scenario which allows us to analyze it, We discuss this in detail in section 4.1

This thesis is structured as follows.

- In **Chapter 2** we survey on the technologies that are relevant for this thesis.
- In **Chapter 3** we present the general description of our approach as well as the design of our prototype.
- In **Chapter 4**, we talk about the most important task that does our framework in detail, that is Generate the Prism code.

- In **Chapter 5**, we describe a use case and Prism code generated by our Framework.
- Chapter 6 concludes the thesis.

2 - Background

In this chapter, we talk about the technologies that are relevant for this thesis.

In particular, we briefly present Domain Specific Languages (DSL) and the Virtual Scenario Description Language (VSDL). Then we will describe the Prism Model Checker and the Discrete-Time Markov Chains (DTMC).

2.1 - Domain-Specific Languages

Domain Specific Languages (DSL) are specification languages that address the domain of a specific problem. In opposition to General Purpose Languages (GPL) (e.g., Java and C), they are not meant to provide features for solving all kinds of problems.

We prefer to create a domain-specific language and not to reuse an existing language. If the DSL enables us to express the problem more explicitly than a general purpose language and the type of problem appears sufficiently often, in these cases, we will be able to solve that problem easier and faster using that DSL instead of a GPL. **Mathematica** (for symbolic mathematics), **SQL** are two examples of DSLs.

2.1.1 - Xtext

Xtext is an Eclipse based framework for implementing programming languages and DSLs. It enables users to quickly implement languages and also takes care of implementing all aspects of a complete language infrastructure including, for instance, linker, compiler, type checker and parser.

Here we use a simple example from the Xtext documentation³ which helps us better understand how Xtext works.

Example 1.

Consider the fragment in Figure 2-1. It represents a simple language for writing entities (e.g. person and address) with attributes that we see in the Figure.

³ <http://www.eclipse.org/Xtext/documentation/2.5.0/Xtext%20Documentation.pdf>

```

Person {
  Name:
  surName:
  birthDay:
  homeAddress:
}

Address {
  street:
  number:
  city:
  ZIP:
}

```

Figure 2-1 Xtext Entity Example

For convenience, first we define the Data types:

```

typedef String
typedef Integer
typedef Date mapsto java.util.Date

```

Grammar:

First, we need to define that our DSL supports data types

```

Model:
  (types+=Type)*;
Type:
  TypeDef | Entity;

```

Figure 2-2 Model Grammar

The two lines that we see in Figure 2-2 define for Xtext that our Model includes any number (i.e. 0..N,) of Types. we need to specify what is a Type, Type can be either TypeDef or an Entity.


```
TypeDef:  
"typedef " name=ID ("mapsto" mappedType=JAVAID)?;
```

Figure 2-3 typedef grammar

As shown in Figure 2-3, A **TypeDef** begins with the keyword `typedef`, succeeded by an ID that denotes its name. Following the name, we can **optionally** (question mark `?` at the end) add a `mapsto` clause. `mappedType=JAVAID` defines that the `TypeDef` will have an **attribute** named **`mappedType`** of type `JAVAID`. we need to define `JAVAID` (figure 2-4)

```
JAVAID:  
name=ID("." ID)*;
```

Figure 2-4 mapsto Grammar

So, a `JAVAID` is a sequence of IDs and dots, such as `java.util.Date`. Now we need to define how to model entities. (Figure 2-5)

```
Entity:  
"entity" name=ID ("extends" superEntity=[Entity])?  
{  
  (attributes+=Attribute)*  
};
```

Figure 2-5 extend Grammar

Entities begin with the keyword `entity`, followed by an ID as their name. They can optionally extends another entity.
[] means "this is a **cross reference**", i.e. a reference to an already existing element. Entities have Attributes so we have to specify how Attributes look like (Figure 2-6)

Attribute:

type=[Type] (many?="*")? name=ID;

Figure 2-6 Cross Reference

An Attribute has a type which is a cross reference to a Type, it has an optional multiplicity indicator and has a name.

In Figure 2-5 we can see the complete grammar:

Model:

(types+=Type)*;

Type:

TypeDef | Entity;

TypeDef:

"typedef" name=ID ("mapsto" mappedType=JAVAID)?;

JAVAID:

name=ID("." ID)*;

Entity:

"entity" name=ID ("extends" superEntity=[Entity])?

"{"

(attributes+=Attribute)*

"}";

Attribute:

type=[Type] (many?="*")? name=ID;

Figure 2-7 total Grammar

2.1.3 - Xtend

Xtend [6] is a general purpose programming language that has complete interoperability with Java entirely developed with Xtext.

Xtend originates from Java programming language, but it has a more concise syntax than Java and some additional functionalities such as ***multi line template expressions***⁴, that are used for writing code generators.

2.2 - Virtual Scenario Description Language [17]

The Virtual Scenario Description Language (VSDL) is a DSL for modeling virtual scenarios for a Cyber Range.

The most important features of this framework are the following.

- **Verifiability**: ensuring that a scenario exposes a specific characteristic, e.g., the presence of a certain vulnerability.
- **Expressiveness**: describing a scenario through a rich syntax, covering many aspects of the scenario.
- **Compositionality**: existing scenarios can be modified and extended by adding new statements to a previous model.

2.2.1 - Scenario in VSDL

A scenario is identified by a name and a sequence of *scenario* elements.

Scenario elements can be nodes or networks. Both nodes and networks are identified by a unique identifier.

For **nodes**, we can express:

- type (e.g., server, client, mobile device, etc.);
- hardware profile (number of CPUs, RAM and Disk size, etc.) ;
- software features (specific operating system or operating system family, e.g., Windows, Linux, Android, etc.).

For **networks**, it is possible to define:

- access to the public network and
- addressing and connected nodes.

⁴ We explain it in the Implementation chapter

Example 2.

Consider the scenario shown in Figure 1-5 and in particular the **Meeting Room** and **Laboratory** areas. In Figure 2-3, we report a possible representation in VSDL.

For **Laboratory and MeetingRoom networks**, the following capabilities are specified:

- direct access to the Internet and
- which nodes are connected to the network.

The mobile phone movement specification from the MeetingRoom network to the Laboratory network is expressed by the guarded constraint at least and at most.

For the Laboratory Network, the following attributes are specified:

- the phone node is connected at most after a time $t = 15$ and
- is in unconnected state for a time value v between 0 and t .

```
scenario Example_Scenario {  
  node laptop1 {  
    disk size equal to 20 GB;  
    OS: "Windows 8";  
  }  
  node laptop2 {  
    not disk smaller than 12 GB and not disk larger than 25 GB;  
    OS: "Windows 10" ;  
  }  
  node fileserver {  
    flavour server;  
    OS: "Linux";  
  }  
  node phone {  
    flavour mobile;  
    OS family: "Android";  
  }  
  network Laboratory {  
    gateway has direct access to the Internet;  
    node laptop1 is_Connected;
```

```

node laptop2 Is_Connected;
[ at most before t = 15 ] -> node phone Is_Connected;
[ at most before v = t and at least after v = 0 ] -> not node phone
Is_Connected
}
network MeetingRoom {
gateway has direct access to the Internet;
node fileserver Is_Connected;
[ at most before t = 15 ] -> not node phone Is_Connected;
[ at most before v = t and at least after v = 0 ] -> node phone
Is_Connected; } }

```

Figure 2-8 VSDL Example

2.3 - Prism Model Checker

Prism is a probabilistic model checker, that is a tool for modeling and analyzing systems that show random or probabilistic behavior. Such systems include, for instance, security protocols, randomized algorithms and more [PRISM].

Models are defined in the Prism Language while properties can be specified through a variety of temporal logics like PCTL, CSL, LTL, and PCTL*. The input formalism for Prism are markov chains (see Section 2.3.1).

Figures 2-9 and 2-10 graphically represent the model checking and probabilistic model checking. Probabilistic model checking allows us to do a Quantitative Analysis of systems .

Quantitative, as well as **qualitative specifications**, are **essential** .for example

- – “ how reliable is my car’s Bluetooth network? ”
- – “ how secure is my bank’s web-service?”⁵

In the Cyber Range field:

- What is the probability that an attack is viable?
- How much can be the probability that an attack will be successful?

⁵<http://www.prismmodelchecker.org/lectures/pmc/01-intro.pdf>

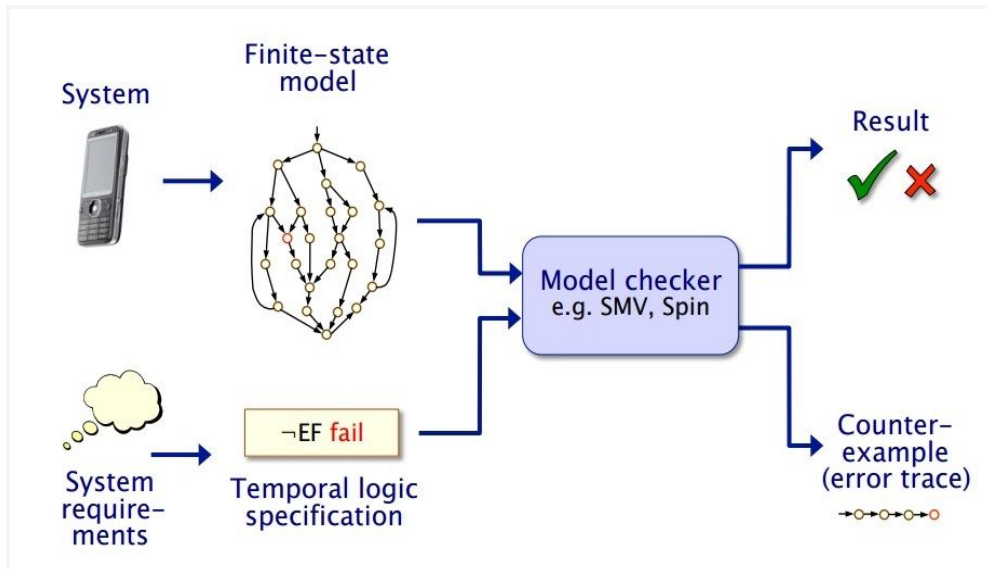


Figure 2-9 Model Checking

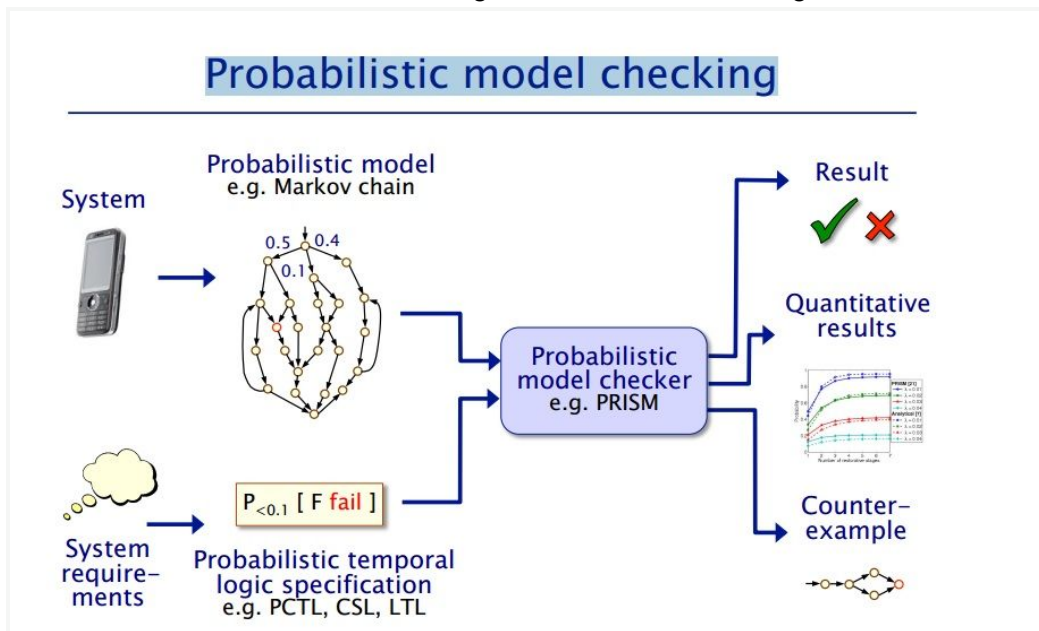


Figure 2-10 Probabilistic Model Checking

2.3.1 - Markov Chains

A markov chain is a type of markov process. A Markov process is defined as “a stochastic process that satisfies the Markov property (sometimes characterized as “memorylessness”). Roughly speaking, a process satisfies the Markov property if one can make predictions for the future of the process based solely on its present state just as well as one could knowing the process’s full history, hence independently from such history; i.e., conditional on the present state of the system, its future and past states are independent.” [18]

A markov process is called a markov chain when its state space is discrete, i.e., finite or countable.

Less formally, markov chains can be seen as sequences of stochastic events where the current state of a variable or system is independent of all past states.

The input models for Prism can be one of the following.

- Discrete-time Markov Chains (DTMCs)
- Continuous-Time Markov Chains (CTMCs)
- Markov Decision Processes (MDPs)

Discrete Time Markov Chain (DTMC)[18]

A stochastic process with discrete state space and discrete time $\{X_n, n > 0\}$ is a discrete time Markov Chain (DTMC) iff

$$P[X_{n+1} = j \mid X_n = i_n, \dots, X_0 = i_0] = P[X_{n+1} = j \mid X_n = i_n] = p_{ij}(n)$$

Continuous-Time Markov Chains (CTMCs)[18]

A continuous-time Markov chain is a stochastic process having the Markovian property that the conditional distribution of the future $X(t + s)$ given the present $X(s)$ and the past $X(u)$, $0 \leq u < s$, depends only on the present and is independent of the past.

Markov Decision Processes (MDPs) [18]

provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

2.3.3 - Prism Modelling Language

To have some idea how the models in Prism Model Checker are defined, we see a simple example ⁶ to explain the basic concepts of the Prism language. Suppose a system including two identical processes which must work under mutual exclusion. Each process has 3 states : {0,1,2}. (Figure 2-11)

In state 0, a process has 2 choices:

- go to state 1 with probability 0.2 or
- remain in state 0 with probability 0.8.

In state 1, the process tries to go to a critical section (state 2) if the other process is not in its critical section.

in state 2 with probability 0.5 can remain here or returns to state 0.

Using another example taken from Prism Official Website, are able to explain two important concepts in Prism model checker(generally for all Probabilistic Model Checker) :

- Parallel composition of modules
- Synchronization

Parallel Composition

In each state of the model, there are some enabled commands (their guards are satisfied), which command is chosen between these commands?

The choice depends on the type of model. For a DTMC, the choice is probabilistic: each enabled command can be selected with the same probability.

In state (0,0) ($x=0$ and $y=0$) (Figure 2-12), the probability distribution is:

$$0.8:(0,0) + 0.1:(1,0) + 0.1:(0,1)$$

⁶ From Official Prism Website

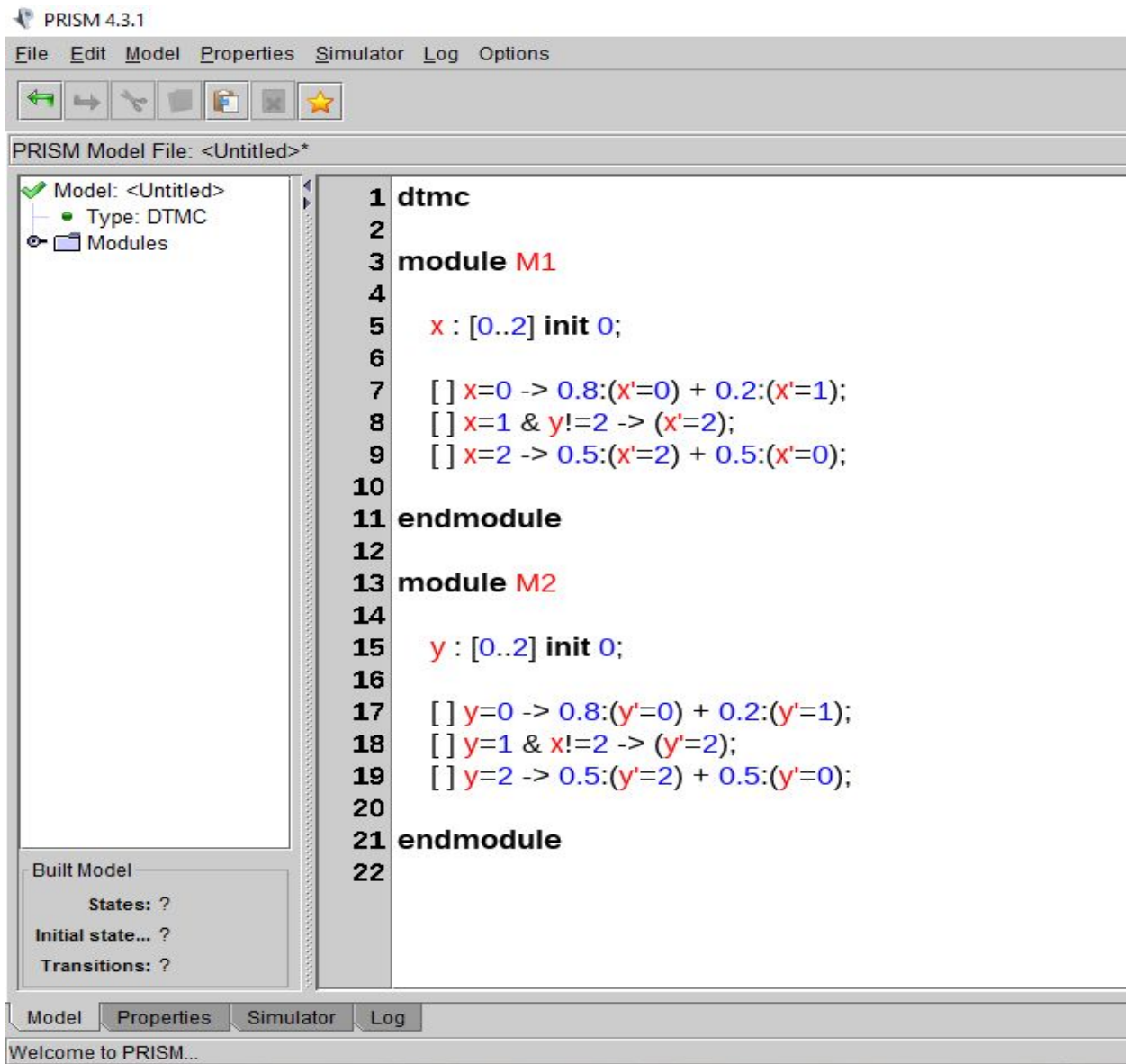


Figure 2-11 Prism Example 1

Synchronisation

In Prism the commands can be labeled with actions. These actions will be used to make two (or more) modules to perform transitions simultaneously .

To explain the concept of synchronization, we use an example taken from the official website of Prism Model Checker:(Figure 2-12)

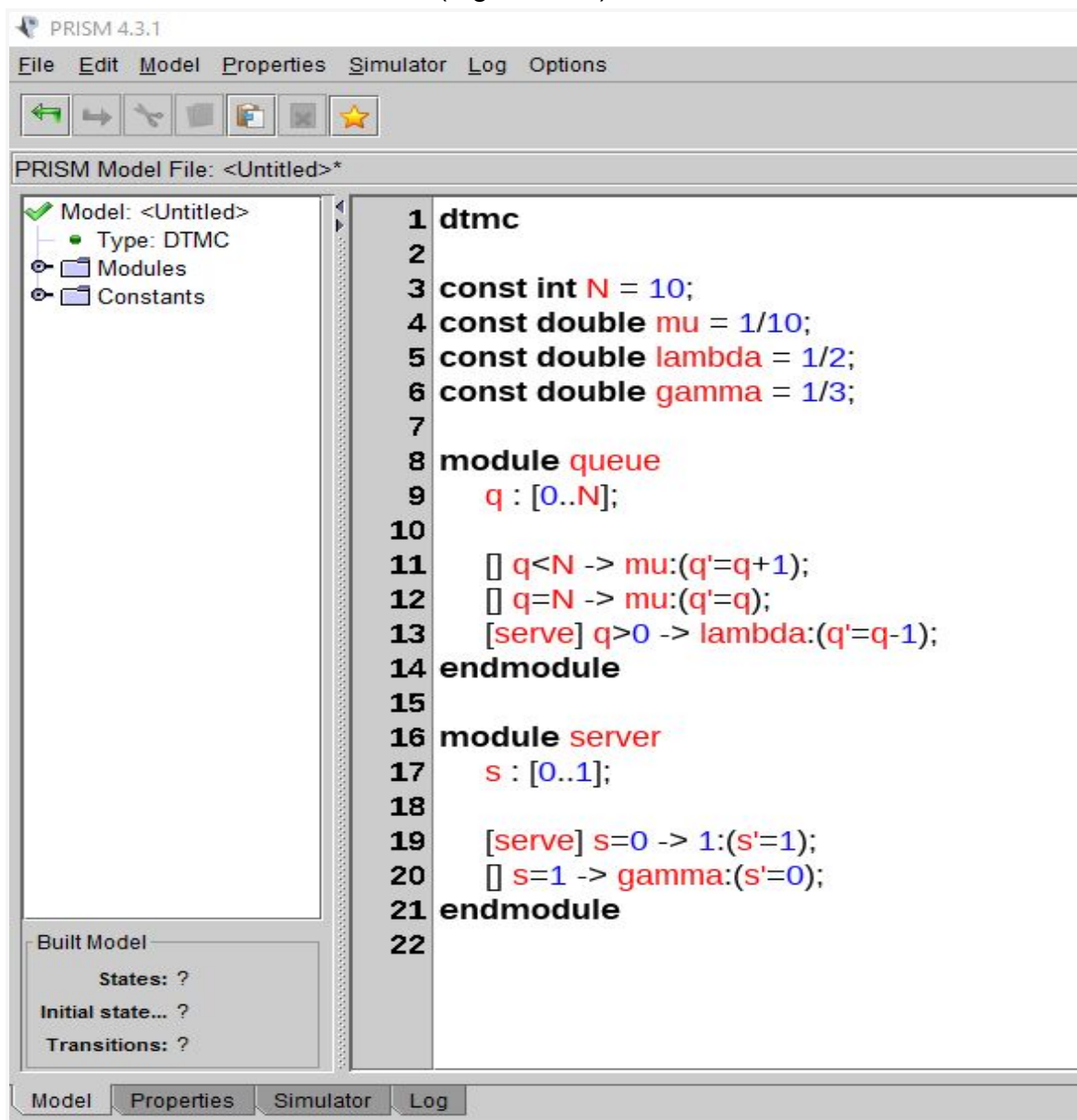


Figure 2-12 Prism Example 2

“A DTMC that models an N-place queue of jobs and a server which removes jobs from the queue and processes them⁷”.

In this example we can see some commands that are labeled with **serve** action

⁷ <http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Example2>

[serve] $q > 0 \rightarrow \text{lambda}:(q'=q-1);$

For example, in the state (3,0) (i.e. $q=3$ and $s=0$), the composed model can go to the state (2,1), synchronizing over the *serve* action.

2.3.3.3 - Prism Property Specification

Prism property specification language includes temporal logics such as PCTL, CSL, LTL and PCTL(Probabilistic CTL). PCTL is used for specifying properties of DTMC. For the goal of this thesis, the most interesting property is Reward Property, PRISM models can be augmented with information about rewards. At first, we talk about rewards in general and then we describe Reward-based Properties. In section 2.3.3.3.3 .

The idea is simple: the probabilistic models written in Prism Language can be augmented with rewards, values associated with some states or transitions of the model. Reward structure can be used to compute properties such as expected score earned. Reward can be assigned to both state and transition of a model, below we see, how would be the reward structure for states and transitions

State Reward

The Reward structure that we see in the Figure 2-9, assigns a reward of 20 to each state of the model. , the left part (true) is a guard and the right part of (20) is a reward. in simple words, the states that satisfy the the guard are assigned the reward.

```
rewards
  true : 20;
endrewards
```

Figure 2-13 State Reward

Transition Reward

Rewards can be assigned to transitions of a model too. These are defined in the same style as state rewards, within the **rewards ... endrewards** construct. Transition rewards are written in the following syntax:

[action] guard : reward;

which can be interpreted as:

“Transitions from states which satisfy the guard **guard** and are labelled with the action **action** acquire the reward **reward**”⁸

For example:

```
rewards
  [ ] true : 200;
  [connected_to_network_A] true : 500;
  [sql_injection] true : 10000;
endrewards
```

Figure 2-14 Transition Reward

It can be interpreted as:

- “assigns a reward of 200 to all transitions in the model with no action label”⁹
- rewards of 500 to all transitions labelled with actions connected_to_network_A
- rewards of 10000 to all transitions labelled with actions sql_injection

Reward-based Properties

We have already discussed the Reward Structure, here we see how the Reward-based Properties are specified. reward-based property can be specified in two ways:

- R **bound** [*rewardprop*]
- R **query** [*rewardprop*]

bound takes the form :<r , <=r , >r or >=r,

query can be one of these: =?, min=? Or max=?.

These can be interpreted as:

- “R **bound** [*rewardprop*] is true in a state of a model if **the expected reward associated with *rewardprop* of the model when starting from that state meets the bound *bound***”
- “R **query** [*rewardprop*] returns the actual expected reward value.”¹⁰

Total Reward Properties

R=? [C]

Returns the expected total number which we have specified in reward structure.

⁸ <http://www.prismmodelchecker.org/manual/ThePRISMLanguage/CostsAndRewards>

⁹ <http://www.prismmodelchecker.org/manual/ThePRISMLanguage/CostsAndRewards>

¹⁰ <http://www.prismmodelchecker.org/manual/PropertySpecification/Reward-basedProperties>

3 - Design

In this chapter we present the general description of our approach as well as the design of our prototype. The design will drive the prototype implementation, as discussed in the next chapter.

First we extend VSDL syntax to include node vulnerability statements (Section 3.2). A vulnerability statement include a pointer to a vulnerability descriptor given through a dedicated scripting language described in Section 3.4. Then we implement a Prism model generator that combines the VSDL specification and the vulnerability descriptors. The result is a model to be verified with Prism.

3.1 - Overall Architecture

Figure 3-1 shows the abstract architecture of our framework. In words

- **Xtext**
- **Xtend**
- **VSDL** is used to write scenarios of Cyber Range
- We have also **JSON** files for the type of vulnerability that is used to create the appropriate Prism Model
- And to have consistent JSON files we have created a specification **Vulnerability Script** for creating the proper JSON file for any type of vulnerability.
- at the end **Model Generator** that creates prism model

From a behavioral perspective, we aim at implementing the logical flow depicted in Figure 3.2. Intuitively, a designer starts by writing scenario in VSDL, and then generates Prism Model from the scenario for verifying vulnerability exploitation conditions, at this point has two choices can accept the scenario or refine it.

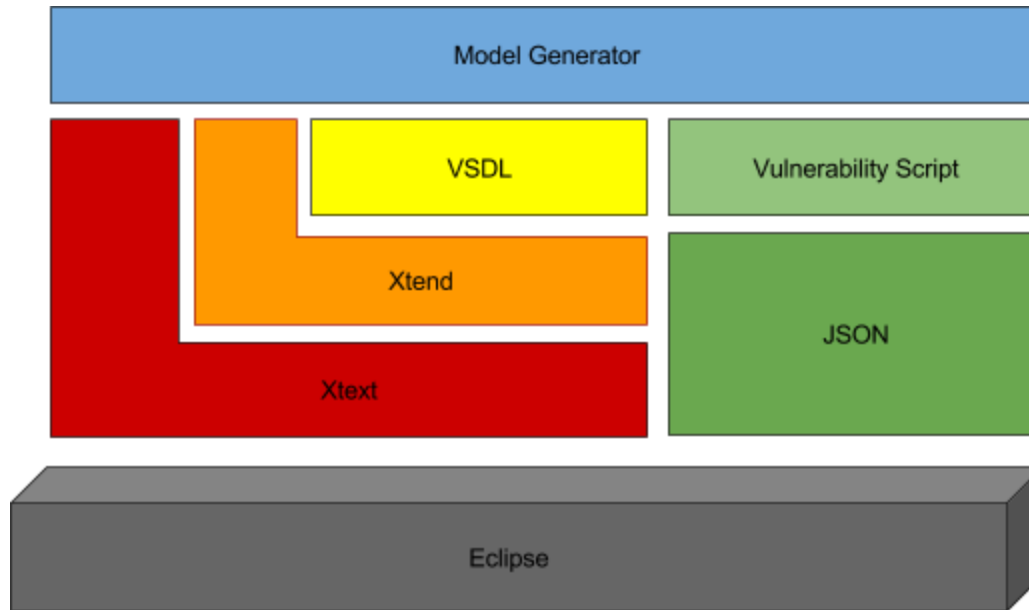


Figure 3-1 Abstract Architecture

3.2 - Vulnerability specification in VSDL

As discussed in Section 2.2, a VSDL specification describes a scenario in terms of its core elements, i.e., nodes and networks, and their features. In this context we see vulnerabilities as extra node features.¹¹ Thus we extended the syntax of VSDL with a new node attribute. The format is

Vulnerability:<identifier>

We propose the following example to clarify.

Example 3.

With an example¹², we try to describe it better:

One of the most common web hacking techniques is SQL injection which is intended to make the database to run unauthorized SQL queries.

¹¹ Although, in principle, also network vulnerabilities might be modeled, we could not identify practical example in the literature.

¹² https://www.owasp.org/index.php/SQL_Injection

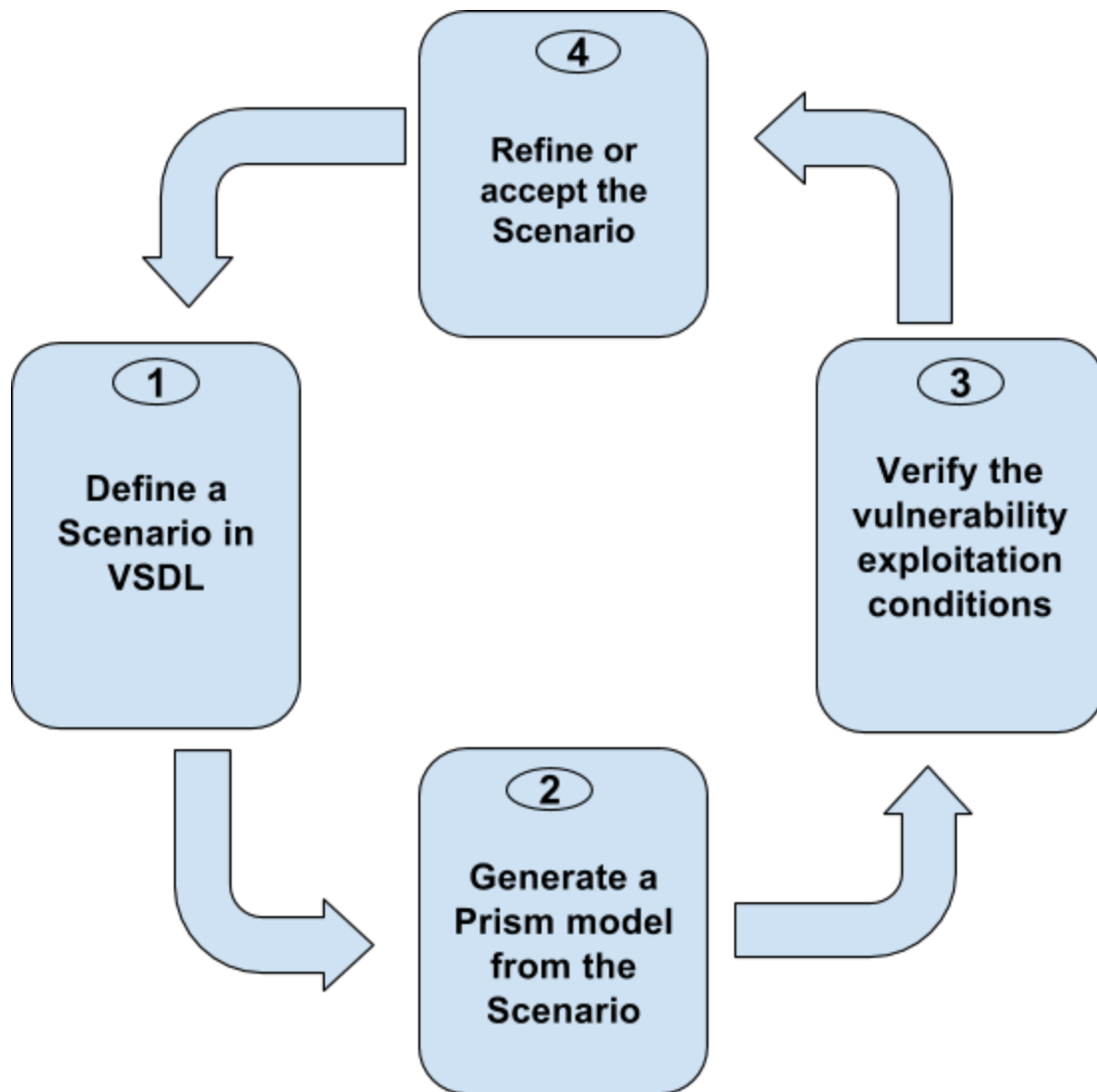


Figure 3-2 Logical Flow

Consider the following query:

SELECT * FROM table WHERE username='\$user' AND password='\$pass'

\$user and **\$pass** are set by the user and suppose no restriction on their syntax is applied.

Let's see what happens by entering the following values :

\$user = ' or '1' = '1 \$pass = ' or '1' = '1

The resulting query will be:

SELECT * FROM table WHERE username=" or '1' = '1' AND password=" or '1' = '1'

The result will always be true, thus leading to an unexpected behavior (e.g., the whole content of table is returned).

So, to study this type of attack, we can specify the vulnerability feature of a node in the scenario .In Figure 3-3 we specified that Pc-1 suffers from Sql-Injection

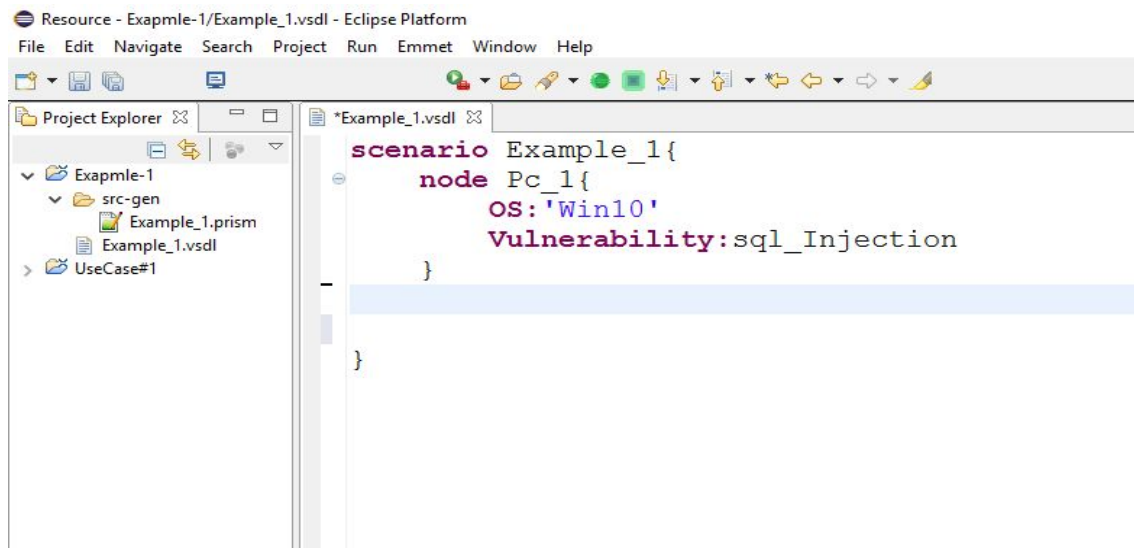


Figure 3-3 sql Injection

3.3 - Code Generation

Until now we talked about how we can specify the vulnerability for a node. In this section we describe the code generation process.

Our framework takes the required information for generating the Prism Code from JSON files. This implies that **for each type of vulnerability**, we need **an appropriate JSON file**.

The whole process can be explained in this way: the Framework reads the type of the required vulnerability from the scenario and then takes the necessary information from the appropriate JSON file to create the Prism code.

For example, if we select the Cross-site Scripting (XSS¹³) vulnerability, we need a file called XSS.json, containing useful information to create the correct Prism Model. the model will obviously be written in Prism language. The main question here can be: how is this JSON file created?

In the next section, we answer this question in details.

3.4 - Vulnerability Specification Scripting Language

We created a specification script for creating the appropriate JSON file for any type of vulnerability.

Each scenario can be represented by a prism model that can contain Global Variables, Constants, some modules and of course the reward structure.

Considering the components of each model and the features of the VSDL scenarios, we created a structure for the JSON file, which can be seen in Figure 3-4.

```
{
  "Head": {
    "GlobalState": [
    ],
    "NetworkConnected": " "
  },
  "Body": {
    "Affected": {
      "state": [],
      "Transitions": []
    },
    "Unaffected": {
      "state": [],
      "Transitions": []
    }
  },
  "Reward": []
}
```

Figure 3-4 Specification Scripting Language

We divided the Json Object into three main parts:

1. Head,
2. Body, and
3. Reward.

¹³[https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))

In the next paragraphs, we explain each of these three parts individually.

3.4.1 - Head

It can include *Global Variables* (all modules can read and write these) and *Constants*. It's made by two parts:

- GlobalState: is an array and can contain *Global Variables* and *Constants*.
For example:
 - *Global known_pc: [0..10] init 0;* or
 - *Const int Budget*
- NetworkConnected: we consider *NetworkConnected* as a *specific constant* to represent whether node X is connected to Network Y or not.

We have to specify it in this way:

```
const bool $nodeName_connected_to_network_$networkName;
```

If we need this information to analyze a type of vulnerability, we have to specify it in the appropriate JSON file that case, the Framework takes and replaces \$nodeName and \$networkName with the correct information. For example:

If the scenario specifies that

pc2 is not connected to network MeetingRoom,

the generated code is:

```
const bool Pc_2_connected_to_network_MeetingRoom= false
```

otherwise, if we don't need these type of information, we can omit it and leave it blank.

3.4.2 - Body

A module is created for each node. If this node hosts a vulnerability it is marked Affected, otherwise, it is marked as Unaffected. States and transitions for a vulnerable node are specified in the Affected Object and the states and transitions of the other nodes are specified in the Unaffected Object.

For example, if node Pc_1 suffers from Injection Vulnerability, the Affected object can be specified in this way:

```

"Affected": {
    "state": [
        "$currentNode_injected:bool init false "
    ],

    "Transitions": [

" [ ]$currentNode_injected=false & Budget>=10 -> ($currentNode_injected'=true);" ,

"$AllOtherNodes_scan]
$AllOtherNodes_connected_to_network_$NetworkCurrentNode=
    true & $currentNode_injected=true -> 1:true; "

    ]
},

```

Figure 3-5 Specification Scripting Language -2

For Pc_1, the framework takes information from Affected Object and replaces \$currentNode with “PC1”, \$AllOtherNodes with all other non-vulnerable nodes and \$NetworkCurrentNode with the network that node pc_1 is connected to. In Figure 3-6 we can see the piece of generated Prism code for Pc_1.

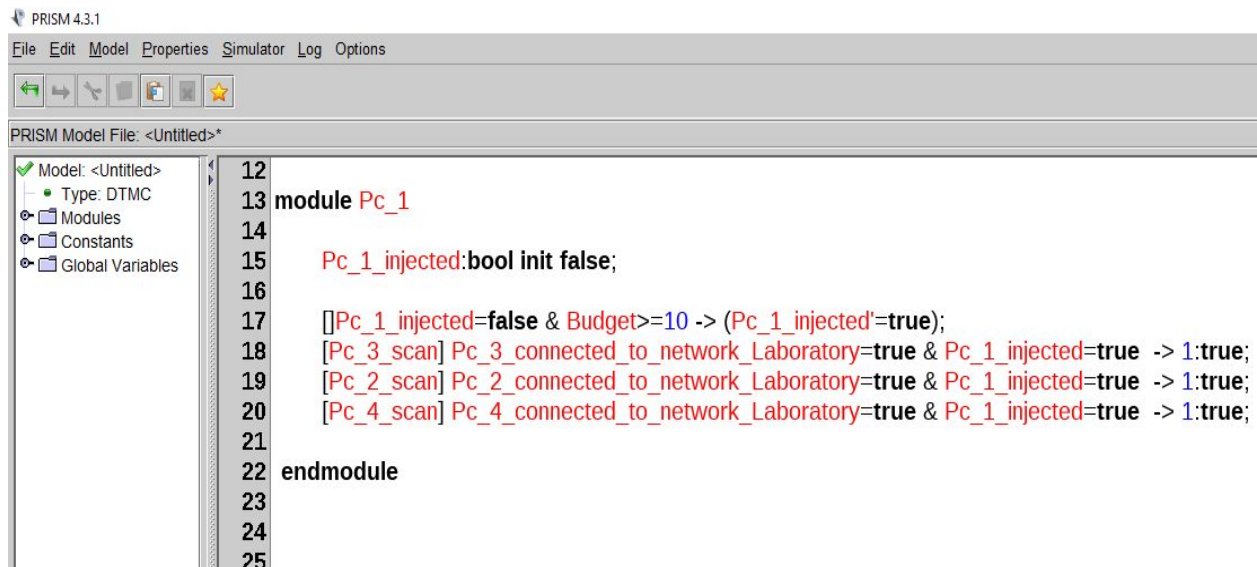


Figure 3-6 Specification Scripting Language -3

3.4.3 - Reward

Is an array that contains the information which is used by the framework for reward structure. for example:

```
[$AllOtherNodes] true : known_pc+1;
```

the framework takes it and replaces `$AllOtherNodes` with all Unaffected node names (Nodes for which there are no vulnerabilities specified in the scenario) we can see the related piece of generated Prism code in figure 3-7.

Previously, we mentioned that we need a JSON file for each type of vulnerability (which is later used by the Framework for generating the Prism code) and we saw how we must create this JSON file.

In this thesis, we have analyzed two types of vulnerabilities¹⁴ and we created two appropriate JSON files for these types of vulnerabilities. By following the structure that we explained above, it's simple to create JSON files for other types of vulnerability.

In the next chapter, we see the implementation details of Code Generator.

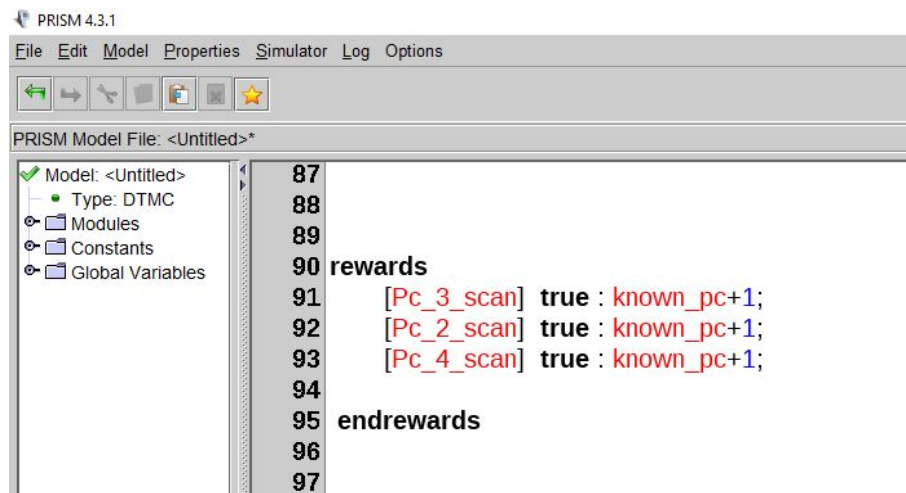


Figure 3-7 Reward

¹⁴ Injection and Command Injection

4 - Implementation

In this chapter, we talk about the most important task that does our framework in detail, that is Generate the Prism code, using information taken from scenarios written in VSDL, this task is done by the Code Generator.

We avoid the details of Eclipse settings, Xtext settings. what we have done to achieve the goal of this thesis can be expressed in a few steps:

1. Create a Xtext project
2. Copy VSDL Grammar in the project that allows us to write scenarios in VSDL.
3. Create vulnerability specification script to insert information that is useful to create the Prism model from our scenario (whether they should be global variables, modules and its states and its transitions) which we explained in detail in the previous chapter
4. Implement the Code Generator that is the core of our project.

4.1 - Writing a Code Generator With Xtend

In the Background chapter, we have already talked about Xtend.

In this chapter, we explain how to write a code generator using the Xtend programming language.

When we create a Xtext project, the code generator is created automatically in Package X.Y.Z.generator (X.Y.Z stand for the domain and the name of our project) in the project. For example in our case, the Generator is created in *it/csec/xtext/hos/generator* package and called *VsdlGeneretor.xtend* as shown in Figure 4-1. The code generator is an Xtend class (as seen in Figure 4-2).

We as programmers don't have to run the generator manually. The generator will automatically be called when our scenario (written in our VDSL) changes. Clearly, the generator will not be called if the scenario has some validation error.

Now we explain how to implement the Code Generator. As we see in Figure 4-2 the *doGenerate* method of the *vsdlGenerator* class has three arguments:

- Resource
- IFileSystemAccess2
- Context

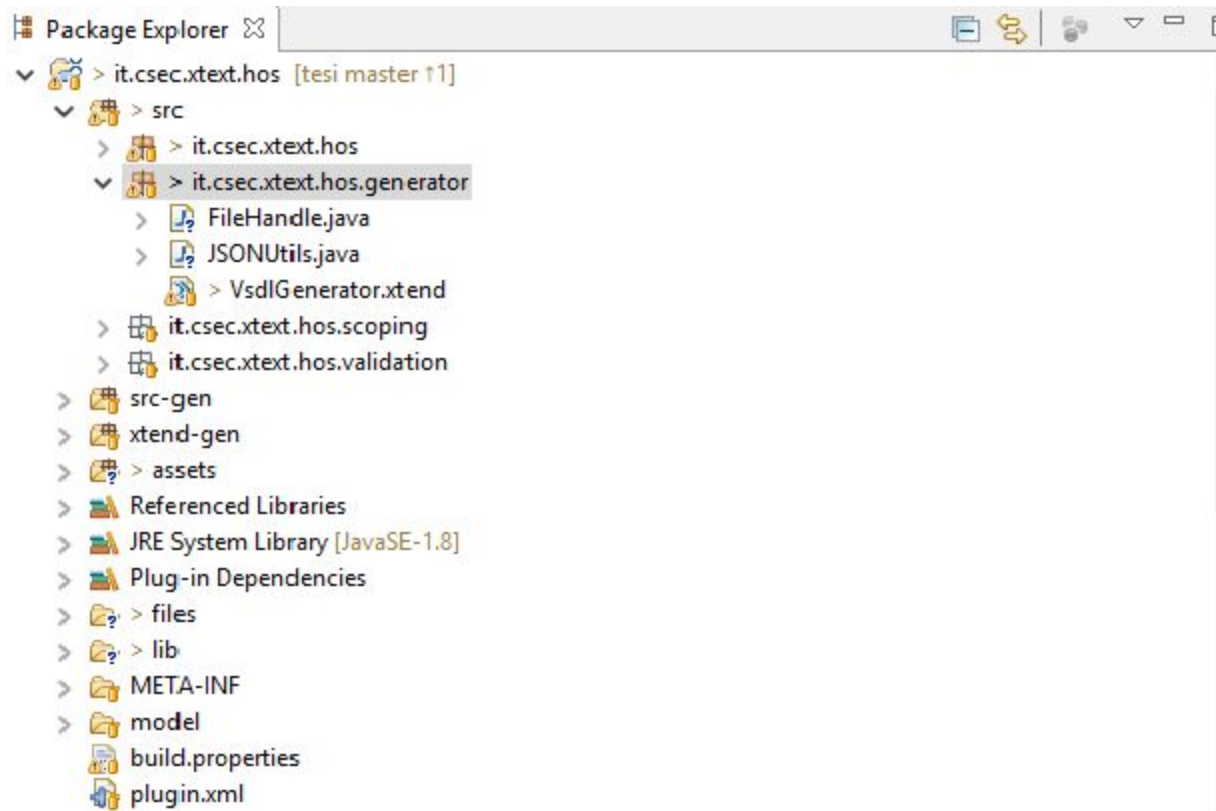


Figure 4-1 Eclipse Code Generator-1

```

Class VsdGenerator extends AbstractGenerator{
    override void doGenerate(Resource resource,
                          IFileSystemAccess2 fsa, IGeneratorContext context) {
    }
}

```

Figure 4-2 VsdGenerator

Now we try to explain how to implement the Code Generator. As we see in Figure 4-2 the *doGenerate* method of the *vsdGenerator* class has three arguments:

- Resource
- IFileSystemAccess2

- Context

Below we see a brief description of Resource and IFileSystemAccess2 which we used for the the our code generator.

4.1.1 - Resource

Usually, **Resource** refers to the model, but in our case, it returns a reference to our scenario (written in VSDL), so we can use it to filter our scenario to get its elements, or the features of said elements. We already know that elements of a scenario can be nodes or networks, so we can parse the scenario and save, for example, the information of a node. Information for a node can be its name, its eventual vulnerability, the network to which it's connected and so on.

For example, we can iterate the whole scenario and save all existing nodes (Figure 4-3) and networks (Figure 4-4) in our scenario.

4.1.2 - File System Access

File System Access (FSA) is the physical base path location where our code (Prism Code) will be generated, so we do not have to worry about the location where our code will be generated.

As it can be seen in Figure 4-5, by using FSA we can call the **generateFile** method that takes **fileName** and **contents** as argument that creates a file named *fileName*.

The created file (fileName) will contain **contents** (charSequence), **contents** is the code that will be generated, in our case is Prism code.

Here we have implemented a method called compileModel. In fact, compileModel method is that gets information from the scenarios and appropriate JSON files and returns the generated codes that will be saved in a file with **fileName** name (Figure 4-6).

```
For (n:resource.allContents.tolterable.filter(Node)) {  
  
    //we can save them in a java collection  
  
}
```

Figure 4-3 Node Filter

```

for(n:resource.allContents.tolterable.filter(Network)) {


    //we can save them in a java collection

}

```

Figure 4-4 Network Filter

```
fsa.generateFile(fileName+".prism",compileModel(networkList,nodeList,nodesConnectedtc
```

 ^A void IFileSystemAccess.generateFile(String fileName, CharSequence contents)

Parameters:

fileName using '/' as file separator
contents the to-be-written contents.

Figure 4-5 Code Generator-5

```

fsa.generateFile(fileName+".prism",compileModel (
    networkList,
    nodeList,
    .....
    .....
))

```

Figure 4-6 fsa

4.1.3 - Implementation of *compileModel* method

compileModel method creates a CharSequence that will be stored in the generated file with *fileName.prism* name. For this purpose we used Xtend multi-line template expressions which is explained in the section 4.1.3.2 .

4.1.3.1 - Xtend Method Declaration

To implement a method in Xtend we must use keyword **def**, the default visibility of a method is public. we can explicitly declare it as being public, protected, package or private.

4.1.3.2 - Template Expressions

Templates allow us to write readable string concatenation. utilizing template expression can span multiple lines by using triple single quotes (""").

In Figure 4-7 we see an example of a method declaration in Xtend with multi-line template.

```
def myMethod(List<String> elements) {  
    ""  
    ""  
}
```

Figure 4-7 Method Declaration

Until now, we've seen how we could write the Code Generator in a general way, in Figure 4-7 we can see the structure of the implementation of our *compileModel*.

```
def compileModel(List<Network> networks,  
    List<Node> nodes,  
    Map<String, Set<String>> nodesConnectedToTheNetwork,  
    Set<String> VulnerableNode,  
    .....)  
{  
    ""  
    ""  
}
```

Figure 4-8 compileModel method

In this chapter, we saw the structure of Code Generator in Xtend Language. In the next chapter we can see how our framework works with a Use case.

5 - Use Case

In this chapter, we will describe a use case and Prism code generated by our Framework. First of all, we represent a type of attack called Injection.

5.1 - Injection Attack

Injection is a major problem in web security. It is listed as the number-one web application security risk in the the OWASP Top 10 of 2017 ¹⁵ (Figure 5-1).

“The injection can cause data loss or corruption, denial of service. It can sometimes lead to complete host takeover.” ¹⁶

“ This type of attack allows an attacker to inject code into a program or query or inject malware onto a computer in order to execute remote commands that can read or modify a database, or change data on a web site.”¹⁷

Some common types of injection attacks:

- Code injection
- Cross-site Scripting (XSS)
- Email (Mail command/SMTP) injection
- OS Command injection

For the use case, we consider the generic injection attack, we don't explain in detail how a system can be vulnerable to this attack.

5.1.1 - Scenario

Consider 4 PCs with names pc_1, pc_2, pc_3, pc_4 .We also have two networks called **Laboratory** and **Meeting Room**

One of the PCs is vulnerable to Injection Attack, in Figure 5-2 we can see one of the possible scenarios.

As soon as we save the scenario, a text file called *useCase.prism* is created in the *src-gen* folder, as shown in Figure 5-2.

¹⁵ https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

¹⁶ https://www.owasp.org/index.php/Top_10_2017-A1-Injection

¹⁷ https://www.ibm.com/support/knowledgecenter/en/SSB2MG_4.6.0/com.ibm.ips.doc/concepts/wap_injection_attacks.htm

In Figure 5-3 we can see the Prism code generated by our framework.

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Figure 5-1 Injection

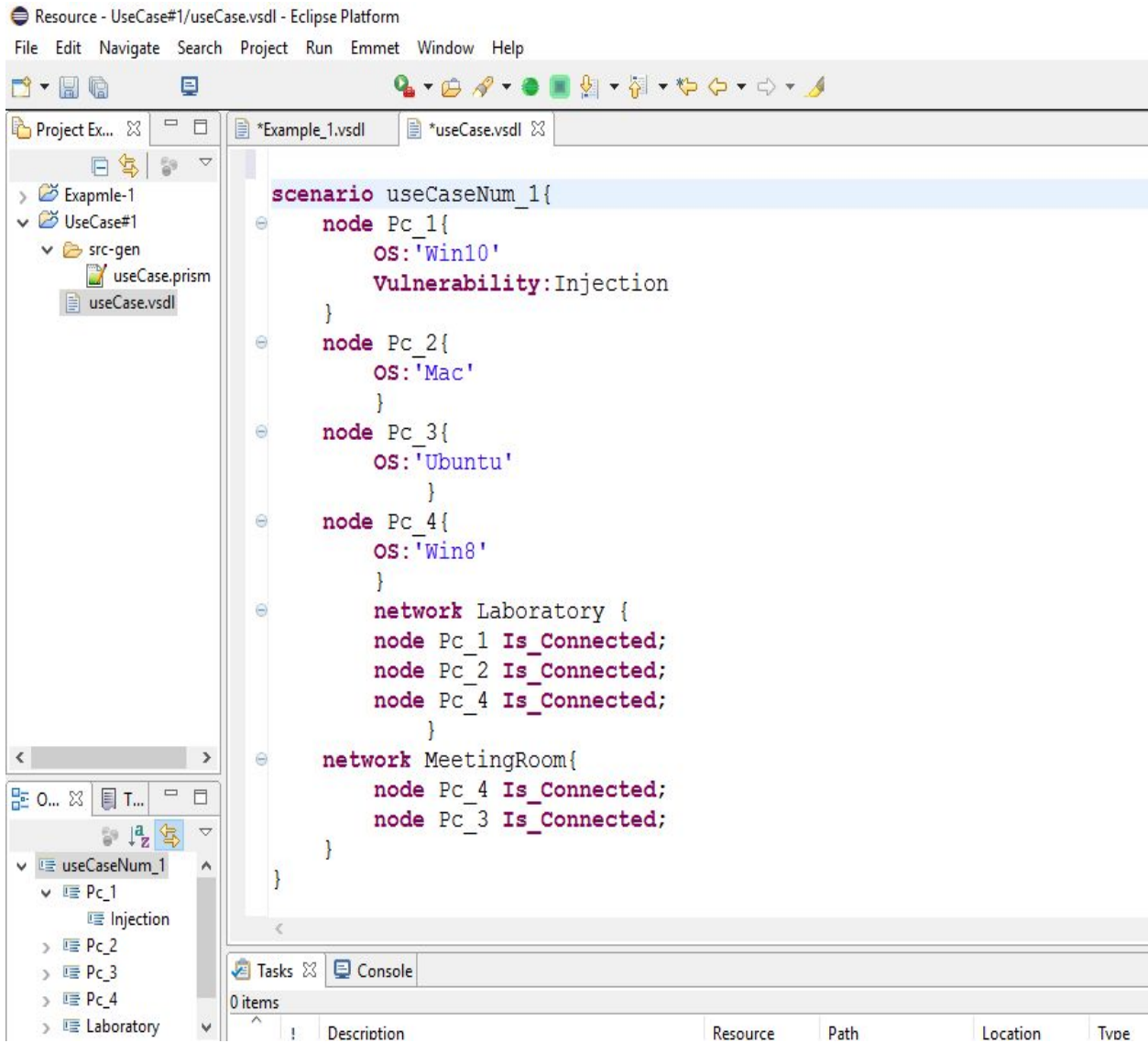


Figure 5-2 Scenario

```

dtmc
global known_pc: [0..10] init 0;
const int Budget;

const bool Pc_1_connected_to_network_MeetingRoom= false;
const bool Pc_1_connected_to_network_Laboratory= true;
const bool Pc_2_connected_to_network_MeetingRoom= false;
const bool Pc_2_connected_to_network_Laboratory= true;
const bool Pc_3_connected_to_network_MeetingRoom= true;
const bool Pc_3_connected_to_network_Laboratory= false;
const bool Pc_4_connected_to_network_MeetingRoom= true;

```

```

const bool Pc_4_connected_to_network_Laboratory= true;

module Pc_1
    Pc_1_injected:bool init false;

    []Pc_1_injected=false & Budget>=10 -> (Pc_1_injected'=true);
    [Pc_2_scan] Pc_2_connected_to_network_Laboratory=true & Pc_1_injected=true -> 1:true;
    [Pc_4_scan] Pc_4_connected_to_network_Laboratory=true & Pc_1_injected=true -> 1:true;

Endmodule

module Pc_3
    Pc_3_scanned :bool init false;

    [Pc_3_scan] Pc_3_scanned=false-> (Pc_3_scanned'=true);

Endmodule

module Pc_2
    Pc_2_scanned :bool init false;

    [Pc_2_scan] Pc_2_scanned=false-> (Pc_2_scanned'=true);

Endmodule

module Pc_4
    Pc_4_scanned :bool init false;

    [Pc_4_scan] Pc_4_scanned=false-> (Pc_4_scanned'=true);
Endmodule

rewards
    [Pc_2_scan] true : known_pc+1;
    [Pc_4_scan] true : known_pc+1;

endrewards

```

Figure 5-3 Prism Code

5.2 Experiment In Prism

Once we have the prism code, we can apply the properties to analyze it. in the Figure 5-3, we can see, for the generated prism model, reward is the **number of Known Pc**, now we can apply for example **Total reward Property** that we explained in section 2.3.3.3.(As seen in the figure 5-4)

In the model we specified that to achieve the attack we need to have a minimum budget of 10, []Pc_1_injected=false & Budget>=10 -> (Pc_1_injected'=true);

Now if we want to verify the property that we see in the Figure 5-5, the prism requires the least budget(Figure 5-5),With Budget 50 we can have the total value of Known pc = 2(As seen in Figure 5-6).

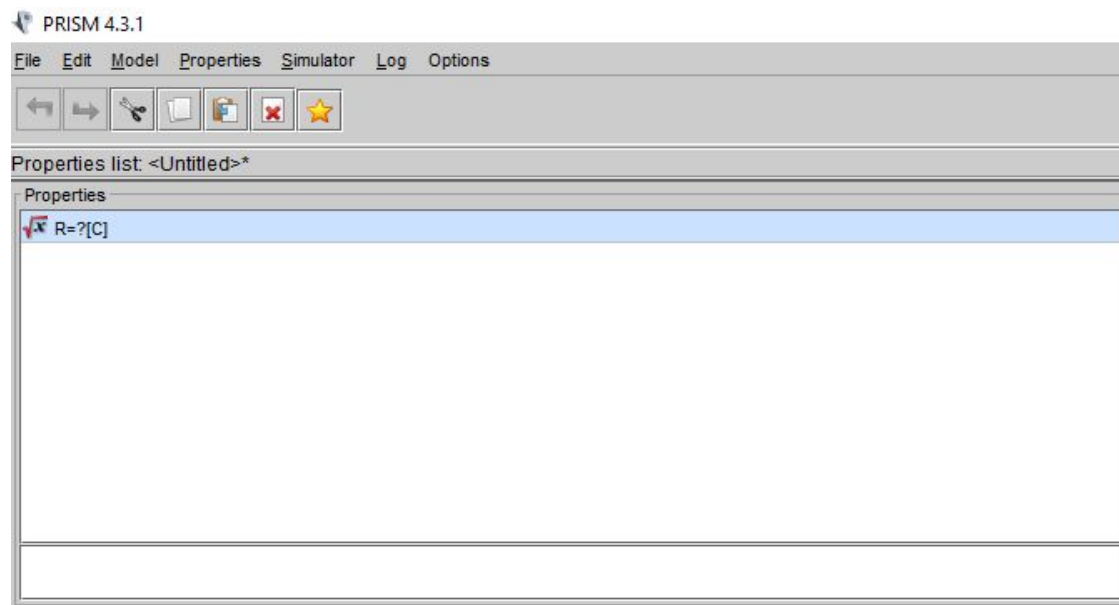


Figure 5-4 Total reward property-1

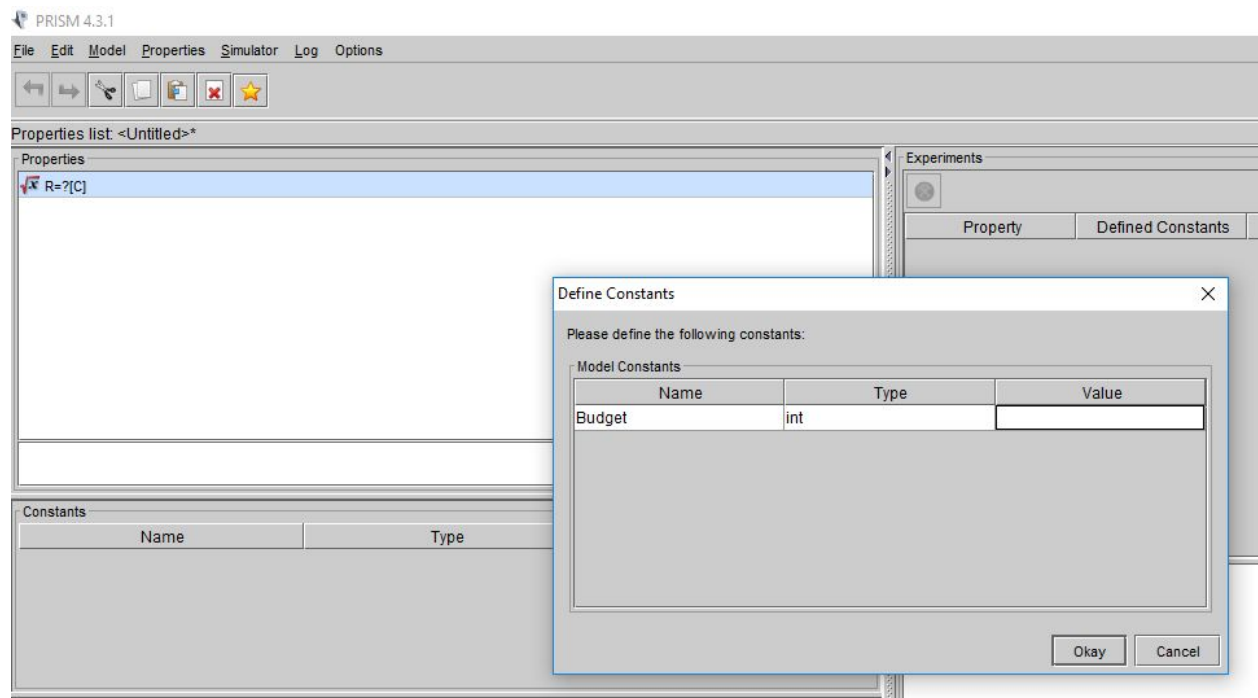


Figure 5-5 Total Reward Property-2

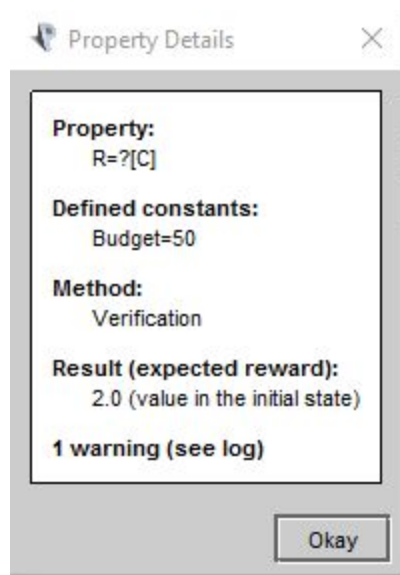


Figure 5-6 Result of property

6 - Conclusion and Future work

In this thesis we implemented a Framework that allows to refine a Cyber Range. In other words, we wanted to design and implement a framework that allows to write scenarios in high-level languages (like VSDL). The framework then creates the Prism code automatically and after that we can apply Reward Property to the created Prism model and use the results to optimize the Cyber Range, to get it closer to a real environment.

The framework parses our scenario and gets the necessary information then loads the appropriate JSON¹⁸ file and with this information can generate the prism model.

We have created a fixed structure for the JSON files that allows us to feed information to the framework to create the appropriate Prism model from our scenario

6.1 - Future work

It would be ideal to have a database of the most famous attacks, at least top ten of OWASP Top Ten¹⁹. Another more interesting thing could be integrating Attack tree into our framework, in this way we can study the attack methods in detail and the relative defense method; we can also study the probability of success of an attack.

¹⁸ associated with the type of vulnerability that was specified in the scenario

¹⁹ Open Web Application Security Project (OWASP)

Bibliography

- [1]OWASP Top 10 2017-Injection, https://www.owasp.org/index.php/Top_10_2017-A1-Injection
- [2] Injection Flaws ,https://www.owasp.org/index.php/Injection_Flaws
- [3]Prism Manual , <http://www.prismmodelchecker.org/manual/>
- [4] Probabilistic Alternating-Time Temporal Logic of Incomplete Information and Synchronous Perfect Recall* , Xiaowei Huang, Kaile Su, Chenyi Zhang
- [5] PRISM-games: A Model Checker for Stochastic Multi-Player Games, Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis
- [6] A theory of timed automata, Rajeev Alur David L. Dill 1994
- [8] Automatic Verification of Competitive Stochastic Systems,Taolue Chen · Vojtěch Forejt · Marta Kwiatkowska · David Parker · Aistis Simaitis
- [9]Book:Implementing Domain-Specific Languages with Xtext and Xtend,Second Edition Lorenzo Bettini
- [10]15 Minutes Tutorial-Xtext,
https://eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html
- [12]15 Minutes Tutorial-Xtend,
https://eclipse.org/Xtext/documentation/103_domainmodelnextsteps.html
- [13]Xtend Extension Method
https://eclipse.org/xtend/documentation/202_xtend_classes_members.html#extension-methods
- [14]Xtend xpressions
https://eclipse.org/xtend/documentation/203_xtend_expressions.html#templates
- [15] Prism Model CHecker Lectures
<http://www.prismmodelchecker.org/lectures/>

[16]Locked Shields: The world's largest cyber-war game
<http://www.aljazeera.com/indepth/features/2017/05/locked-shields-world-largest-cyber-war-game-170527102554714.html>

[17]Creating a framework for the automatic generation of virtual scenarios for Cyber Range.
Enrico Russo 2016

[18] Basics of Applied Stochastic Processes. Richard Serfozo (24 January 2009)

Appendix A - VDSL syntax

grammar it.csec.xtext.Vsdl **with** org.eclipse.xtext.common.Terminals **hidden**(WS, ML_COMMENT, SL_COMMENT)

generate vsdl "http://www.csec.it/xtext/Vsdl"

Model:

'scenario' name=ID (**'duration'** TimeToUse=TimeInterval)? **{** Elements += ScenElem* **}**;

ScenElem:

Node |

Network;

Node:

'node' name=ID **{** constraints += NodeConstraint* **}**;

NodeConstraint:

GuardedNodeConstraint **;** |

SimpleNodeConstraint **;**;

GuardedNodeConstraint **returns** *NodeConstraint*:

{ triggerconstraint=UpdateTriggerConstraint **}** **'->'**
nodeconstraint=SimpleNodeConstraint;

SimpleNodeConstraint **returns** *NodeConstraint*:

SimpleNodeConstraintAndOr;

SimpleNodeConstraintAndOr **returns** *NodeConstraint*:

```
SimpleNodeConstraintNot  
(({AndOr.left=current} op=("and"|"or"))
```

```
right=SimpleNodeConstraintNot)*;
```

SimpleNodeConstraintNot *returns NodeConstraint*:

```
{Not} =>"not" constraint=SimpleNodeConstraintA | SimpleNodeConstraintA;
```

SimpleNodeConstraintA *returns NodeConstraint*: "(" SimpleNodeConstraint ")" |
NodeHardwareConstraintA | NodeSoftwareConstraintA;

NodeHardwareConstraintA *returns NodeConstraint*:

```
{VCPU} =>'cpu' 'number' op='equal' 'to' value=INT |
```

```
{VCPU} =>'cpu' op='greater' 'than' value=INT |
```

```
{VCPU} =>'cpu' op='less' 'than' value=INT |
```

```
{VCPU} =>'cpu' 'number' sameas ?='of' id=[Node] |
```

```
{Disk} =>'disk' 'size' op='equal' 'to' value=DiskSize |
```

```
{Disk} =>'disk' op='larger' 'than' value=DiskSize |
```

```
{Disk} =>'disk' op='smaller' 'than' value=DiskSize |
```

```
{Disk} =>'disk' 'size' sameas ?='of' id=[Node] |
```

```
{Ram} =>'ram' 'size' op='equal' 'to' value=RamSize |
```

```
{Ram} =>'ram' op='larger' 'than' value=RamSize |
```

```
{Ram} =>'ram' op='smaller' 'than' value=RamSize |
```

```
{Ram} =>'ram' 'size' sameas ?='of' id=[Node] |
```

```
{Flavour} =>'flavour' profile=STRING;
```

NodeSoftwareConstraintA *returns NodeConstraint*:

```
'node' {OS} =>'OS' 'is' version=OSVersionE |
```

```
'node' 'OS' {OSFamily} =>'family' 'is' family=OSFamilyE | 'node' 'mounts' {Software}  
=>'software' software=STRING;
```

Network:

```
'network' name=ID '{' constraints += NetworkConstraint* '}';
```

NetworkConstraint:

```
GuardedNetworkConstraint ';|
```

```
SimpleNetworkConstraint ';';
```

GuardedNetworkConstraint *returns NetworkConstraint:*

```
'[' networktriggerconstraint=UpdateTriggerConstraint ']' '->'  
networkconstraint=SimpleNetworkConstraint;
```

SimpleNetworkConstraint *returns NetworkConstraint:*

```
SimpleNetworkConstraintAndOr;
```

SimpleNetworkConstraintAndOr *returns NetworkConstraint:*

```
SimpleNetworkConstraintNot
```

```
(({AndOr.left=current} op=("and"|"or"))  
right=SimpleNetworkConstraintNot)*;
```

SimpleNetworkConstraintNot *returns NetworkConstraint:*

```
{Not} =>"not" constraint=SimpleNetworkConstraintA |
```

```
SimpleNetworkConstraintA;
```

```
SimpleNetworkConstraintA returns NetworkConstraint: "(" SimpleNetworkConstraint ")" |  
NetworkGatewayConstraint | NetworkParticipantsConstraint |  
NetworkFirewallConstraint;
```

NetworkGatewayConstraint *returns NetworkConstraint:*

{IPRange} =>"addresses" "range" "is" range=IPRangeA |

{Gateway} =>"gateway" "has" "direct" "access" "to" "the" internet ?="Internet";

NetworkParticipantsConstraint **returns** *NetworkConstraint*: {IP} =>"node" id=[ScenElem]
"is" op="connected" |
{IP} =>"node" id=[ScenElem] "has" op="IP" address=IPAddress;

NetworkFirewallConstraint **returns** *NetworkConstraint*:

"firewall" "blocks" "port" port=INT |

"firewall" "forwards" "port" sPort=INT "to" dPort=INT | "firewall" "blocks" "IP"
address=IPAddress |

"firewall" "translates" "IP" srcAddress=IPAddress "in"
dstAddress=IPAddress;

UpdateTriggerConstraint:

UpdateTriggerConstraintAndOr;

UpdateTriggerConstraintAndOr **returns** *UpdateTriggerConstraint*:

UpdateTriggerConstraintNot
(({AndOr.left=current} op=("and"|"or"))

right=UpdateTriggerConstraintNot)*;

UpdateTriggerConstraintNot **returns** *UpdateTriggerConstraint*: {Not} =>"not"
constraint=UpdateTriggerConstraintA | UpdateTriggerConstraintA;

UpdateTriggerConstraintA **returns** *UpdateTriggerConstraint*:

(" UpdateTriggerConstraint ") |

{At} =>"at" op="least" "after" variable=ID "=" texp=TimeExpr | {At} =>"at" op="most"
"before" variable=ID "=" texp=TimeExpr | {Switch} =>"switch" "after" variable=ID "="
texp=TimeExpr;

TimeExpr:

TimeExprAddition;

TimeExprAddition **returns** *TimeExpr*:

TimeExprMultiplication

((*{PlusMinus}*.left=**current**) op=("+|-"))
right=TimeExprMultiplication)*;

TimeExprMultiplication **returns** *TimeExpr*:

TimeExprA

((*{Multiplication}*.left=**current**) op='*')
right=TimeExprA)*;

TimeExprA **returns** *TimeExpr*:

(' ' TimeExpr ') | variable=ID | interval=TimeInterval;

TimeInterval:

value=INT;

CPUFrequency:

value=INT unit=('MHz' | 'GHz' | 'THz');

DiskSize:

value=INT unit=('MB' | 'GB' | 'TB');

RamSize:

value=INT unit=('MB' | 'GB');

IPAddress:

octet1=INT'. 'octet2=INT'. 'octet3=INT'. 'octet4=INT;

IPRangeA:

address=IPAddress '/' bitmask=INT;

OSVersionE:
STRING;

OSFamilyE:
STRING;