Formal Methods and Tools (FMT) research group
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

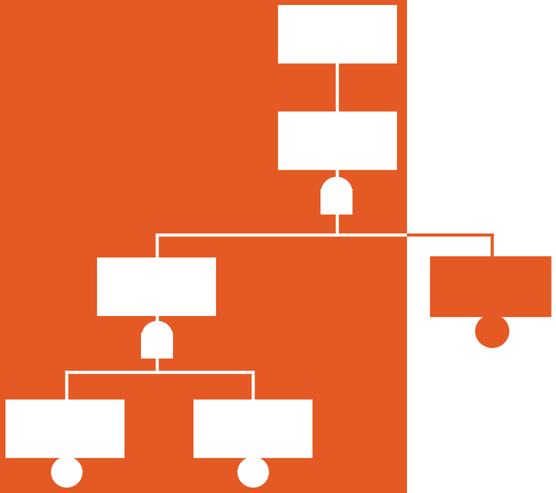
Master thesis

Analysis of Attack Trees with Timed Automata

transforming formalisms through metamodeling

Author:
Wolters N.H.

Committee:
dr. M.I.A. Stoelinga
dr. S.Schivo
R.Kumar, Msc



March 24, 2016

UNIVERSITY OF TWENTE.

Acknowledgements

First of all I would like to express my gratitude towards my committee.
Marielle Stoelinga for her guidance and patience during the graduation process. For always showing interest when presented with updates and for asking the right questions that led to new insights. *Rajesh Kumar* for his always upbeat attitude and willingness to share his extensive knowledge within the *Attack Trees* domain. *Stefano Schivo* for his quick rounds of feedback and his humor within these notes, that made the whole feedback process enjoyable.

Secondly I would like to thank *Daniel*, *Marcus* and *Tara* for proofreading this document, both during my graduation process and in the final weeks leading up to the defence.

Thanks to *David*, whose fruitful collaboration on the whole meta-modelling and *Epsilon* transformation part resulted in very good results.

Thanks to *Tara*, for her always positive attitude and her decisiveness to push for the importance of, and her experience in, creating a design.

Finally, last but definitely not least, for their unquestionable support during last years hard times, I would like to take this opportunity to thank *Jos*, *Femia*, *Tara*, *Noortje*, *Thom*, residents and former residents of *Huize Badeendt*, *family*, *colleagues* and whoever I might have forgotten here.

Abstract

This thesis considers Attack Trees, one of the most prominent security formalisms for analysing threats, and provides methods to transform this formalism into Timed Automata. Which can be analysed with Uppaal, resulting in different types of analysis and thus in additional possibilities for results. This thesis provides a meta-modelling approach to the transformation of input ATs from *ADTool* into an instance of Uppaal. For this to work, we also introduce the *Attack Tree Meta Model (ATMM)* which can be used to store a wide variety of Attack Tree related formalisms. Furthermore the *Epsilon Framework* is used to provide transformations from and towards the ATMM. The *Epsilon Transformation Language (ETL)* is used for model-to-model transformations and the *Epsilon Generation Language (EGL)* for the model-to-text part. This thesis contains the following transformations: *ADTool2ATMM* and *ATMM2ADTool* as ETL model-to-model transformations and *ATMM2Uppaal* as an model-to-text transformations. As a final deliverable all the transformations and internal ATMM are included in a standalone java tool, which can transform ADTool input into a Uppaal model which can be used for Attack Tree analysis.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

We live in a time where we are being, arguably already have been, surpassed by technology. Systems become larger, more complex and at the same time more safety-critical every day. Our society depends on these systems to function correctly as expected and have an 24/7 uptime, but often lack the proper knowledge and skill to ensure this. To verify that these systems cannot 'fail', great progress has been made by research to make the best fault tolerant digital systems, make it stable, use the best resources, and test it thoroughly. Commonly neglected is the possibility of a possible malicious user; a user who will use and abuse every weakness he or she can find in a system.

For example, we can ensure that a 12 letter camel-case password with a few numbers and symbols is potentially uncrackable, but when this password cannot be remembered by a person and he or she chooses to write it down, this becomes a new security risk. A malicious user would in this case be much more likely to find the paper than attempt to crack the password.

Another instance surrounds online bank transactions. A system might use a password and an email message to identify the user. However this system could neglect the possibility of a so called *man in the middle attack* [14], in which case the malicious user can intercept both the outgoing password and the returning email to abuse them at will.

This kind of behavior should be considered when developing new or maintaining existing software systems. For this the European project *TREsPASS* [4] is going to provide an *"attack navigator"*, which will be used to identify weak points and provide possible attack routes and their attached counter-measures to prevent these attacks. This navigator uses multiple steps to get to its final result, one of these steps is the analysis on one of the intermediate models, named *Attack Trees*.

Attack Trees (ATs) are a graphical representation of a combination of low level hostile activities, which when combined make up an attack. Depending on how these low level activities (*basic elements*) are linked to the root of the tree, an attack is either successful or not. The root contains the actual goal for the attacker and is often related the most valuable things in an organization.

The field of attack trees will be the main concern of this work. Attack trees are a formalism which can contain all information that is relevant for large security questions, but extracting useful data out of this has proven to be time-consuming for larger trees. Tools like *ATCalc* [2] and *ADTool* [31][1] are already capable of attack tree analysis, but all have their own limitations or restrictions. *ADTool* for instance can only handle singular attributes at a time, while it might be desired to draw conclusions about multiple.

This thesis provides an Uppaal model for multi-parameter attack trees, a complete meta-model for general attack trees, transformations from and to the meta-model and finally a separate transformation from an Uppaal meta-model into actual Uppaal code. These model transformations from different sources are included, not only for completeness, but also as a proof of concept for future implementations. As outgoing transformations two implementations are provided, firstly one from the meta-model to *ADTool* [31][1] and secondly one from the meta-model to *Uppaal* [5][11], the latter being the major contribution of this thesis, both the resulting Uppaal model and its transformation.

1.1 Background

TREsPASS

TREsPASS [4] is a European project structured around combining expertise from technical sciences, social sciences and state of the art processes and tools to tackle the grand problem of information security threats. The project consortium has partners from different fields who combine their knowledge in a large chain of processes which will be available through the TREsPASS *attack navigator*. This navigator allows for the user to specify an environment, e.g. the layout of a company, together with its employees and their access levels. By analyzing this *socio-technical model* the navigator creates attack scenarios, which are in turn stored in *Attack Trees*.

Attack Trees

Attack Trees allow for both a good visualization and multiple analysis methods. In essence an attack tree has a reversed tree structure which represents a possible attack on a system. The top node is called *root* and is the goal of the attack, e.g. stealing data or preventing access for regular users. Every refinement divides the attack in smaller attack steps, finally ending up with *basic attack steps* (i.e. *cracking a password*, *breaking a lock*, etc). There are different kinds of refinements, two of the most basic are AND and OR refinements. The AND needs all its children to fire before it continues and the OR only needs one. There are more possible refinements, but these will be discussed further along this thesis. In Section 2.4 we will provide some examples of *Attack Trees*.

Attack Tree Analysis

Attack Trees are also very useful for security analysis. Because they provide a formal, methodical way of describing scenarios which can be used for both qualitative and quantitative analysis [43]. Examples of results gathered from these analysis can be used for *Real-Time Monitoring* or *Anomaly Detection* [43]. The analysis can be performed given certain parameters in the leaves. Leaves can contain values like e.g. *cost*, *rewards*, *risk* or *success probability*. Based on how they are linked through intermediate nodes the values of the parameters propagate upwards. For example, if two leaves have a 50% probability of success and they are connected with an AND-gate, the probability of this sub-tree succeeding is 25%. Ways to analyze are numerous, simple trees can easily be checked by hand, while for more advanced trees the use of analytical tools is desired. One of these tools, primarily focusing on real-time systems, is *Uppaal*.

Uppaal

Uppaal is a tool developed by the *Uppsala University* and the *Aalborg University*. This tool is primarily used for the modeling of real-time systems and therefor finds its basis in the form of timed automata. *Uppaal* is a tool that uses the parallelization of these automata, is primarily used in the scientific community and is at the base of a number of different variants, which all allow for different types of analysis. Newer versions of *Uppaal* allow for *Statistical Model Checking* (SMC)[17] or for *Cost Optimal Reachability Analysis* (CORA)[6]. For this research the focus will be primarily on the functionality of standard *Uppaal*, combined with some of the statistical extensions. Since the Attack Trees cannot be directly modeled into an Uppaal model, this thesis has a big focus on the transformation process. For this process, Attack Trees are first transformed into an intermediate model (meta-model), before the second step will complete the transformation to Uppaal.

Meta-modelling

A meta-model or surrogate model is a model of a model. It is to be considered one abstraction higher than general models. It contains multiple entities, rules, constraints and frames which all apply to the instance of this meta-model. In this thesis a meta-model for *Attack Trees* is presented, which is complete enough to represent the majority of the threat formalisms defined by Kordy et al. [32]. The actual *attack tree meta-model* can be found in Appendix F. Using this as an intermediate model should allow future extensions to easily adapt and use the already existing transformation into the *Uppaal-model*. An example of the overall transformation process can be found in Figure 1.1.

A model transformation is the process of converting one model into a different one. This can be done by multiple methods, one of which uses *Epsilon* [28]. *Epsilon* is a modeling framework which was primarily created for *code-generation* and *model-to-model* transformations, but also supports options for *validation*, *comparison* and *refactoring*.

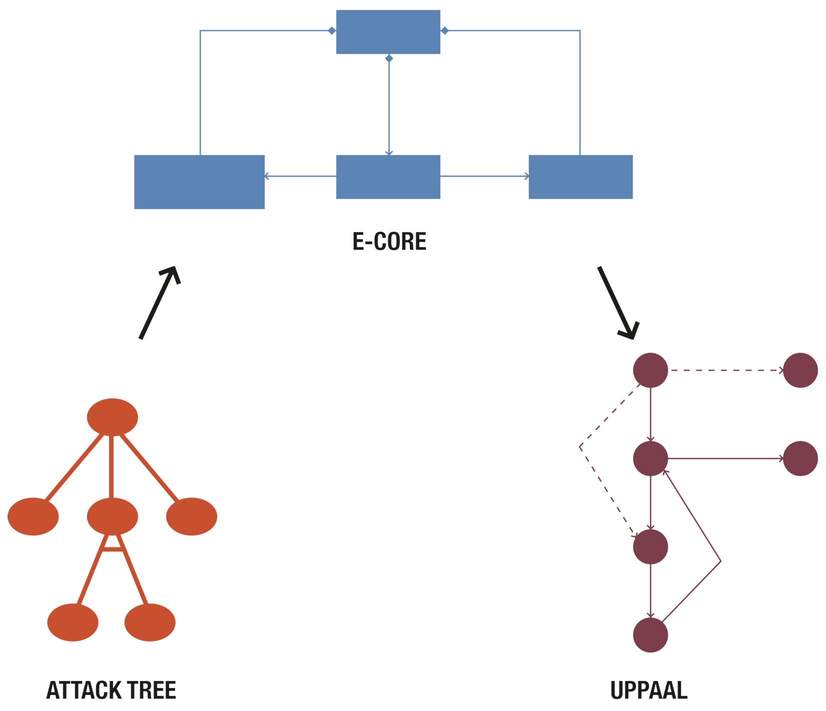


Figure 1.1: Transforming ATs into a Uppaal model, via a meta-model

One of the languages supported by the Epsilon framework is the *Epsilon Transformation Language* (ETL), this rule based *model-to-model* transformation language is a very complete language. It supports multiple input models, output models, rule-types and can be used to access external libraries. A second language relevant to this thesis is the *Epsilon Generation Language* (EGL), this is a *model-to-text* language and is used for creating the Uppaal model. A sub-goal of this thesis is not only to provide transformations from and to our meta-model, but also make it easy for follow-up research to create their own transformations. Included with this thesis are a couple of *ETL-transformations* and one *EGL-transformations*, which can be found in appendices A-D.

Currently the amount of information that can be extracted from *Attack Trees* is limited and there are no real applications that combine ATs with *timed automata*. As extracting this information is a vital step in the TREsPASS attack navigator chain, this issue needs to be fully explored and resolved. Since the current tools can easily calculate static attack trees without combining different parameters, the primary problem that remains is the combination of multiple attributes within a tree and the realistic representation of time in the analysis of *Attack Trees*.

1.2 Deliverables

This thesis achieved the following results.

1. *A multi-parameter Attack Tree model in Uppaal.* Based upon the state of the art, this thesis delivers an Uppaal model which can handle *Attack Trees* with multiple domains of parameters.

2. *An Attack Tree Meta-Model (ATMM).* Abstracting the multiple tools into a more generic intermediate model should allow future extensions to easily adapt and use the already existing transformations.

3. *Multiple transformations are included.*

   (a) *ADTool2ATMM.* A transformation in ETL from the ADTool xml output into an instance of the meta-model.

   (b) *ATMM2ADTool.* Inverse of the previous transformation, also using ETL.

   (c) *ATMM2Uppaal.* This uses EGL to transform an instance of the meta-model into a working Uppaal-model.

   (d) *UppaalEcore2Uppaal.* Separate from our meta-model and the other translations, the need for an additional transformation arose. This transformation requires an Uppaal meta-model instance as its input and converts it to an Uppaal model.

4. *A Tool* incorporating the transformations, one generic tool which can be loaded with the different transformations.

   (a) Version 1. Uses *ADTool2ATMM* and *ATMM2Uppaal*, this delivers a chain of transformations which takes *ADTool* as an input and an Uppaal-model as its output.

   (b) Version 2. Only has the *UppaalEcore2Uppaal.* Takes an instance of the meta-model as input and produces the Uppaal model.

1.3 Document Structure

This document is structured according to the multiple phases in the design process. After the introduction its primary focus lies on *Attack Trees*, what are they, how can they be useful. This will be the content of **Chapter 2**. **Chapter 3** shows how these ATs can be correctly represented in an Uppaal model and will contain the Uppaal templates used for various elements of Attack Trees.

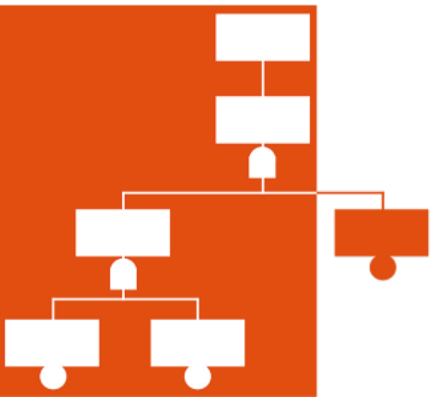
Chapter 4 describes the design and implementation of the *Attack Tree Meta-Model*, why there is a need for a meta-model in the first place and what design choices have been made in the process to creating this meta-model.

In order to make the work more complete, this thesis also presents multiple transformations to and from the meta-model. **Chapter 5** first introduces the transformation options and explains the choices made, followed by the actual transformations and their design choices / limitations. At the end of this chapter both the meta-model and the transformations will be combined in a stand-alone application which can be used without the development environment.

Finally **Chapter 6** will be used to illustrate a transformation demonstration. Starting at an *Attack Tree*, modeling that in *ADTool*. Secondly using this instance to transform it into the meta-model and finally transform it into the Uppaal model.

This concludes the design process and allows us to present the final results, discussion and future work in **Chapter 7**. Finally Chapter 8 will contain the Conclusion.

Chapter 2
Attack Trees



2.1 How we are using Attack Trees

In order to perform *Uppaals* analysis of *Attack Trees (ATs)*, our final goal is to create a tool which can transform an AT into a Uppaal model, hence a stable goal should be specified. This Uppaal model can be used by security analysts to extract the required information. Currently several different versions of the AT-formalism are used in the field, therefore an analysis of the most relevant approaches will provide better insight into what is needed and what the transformation should work towards.

Since our starting point uses the formalized Attack Trees used within the tool *ADTool*, the formalism used in this tool, *ADTrees*, will be the basis of our implementation. The formalism itself has a great structure, is clear and is capable of the most important features of an AT, however a couple of limitations also emerge. ADTrees, only allows for the analysis of single attributes. For every type of attribute (e.g. *cost*) every leaf should have a value of this type. All these similar attributes are combined in one *Domain*. A *Domain* does not only contain the attributes and its types, but most importantly contains the methods on how to calculate these attributes when they encounter AT-gates (e.g. when two *Cost* attributes they are added, while *risk* attributes might be multiplied). And even though the basis of these attributes can be different, there are currently only four types (*satisfiability, time, cost,* and *risk*) available in *ADTool*. These types can be interpreted by the user as they please, but still lacks a certain flexibility. Regarding the time domain, *ADTool* is limited over time abstract analysis, whereas using probability distributions would be far more useful and allow for more analysis.

The final resulting Uppaal model will be able to analyse the same features of ATs as the attackers part of *ADTool*, but combining all the domains into one analysis. This allows for analysing multi-parameters Attack Trees, e.g. combining *costs* with *damage* in questions like *What is the damage an attacker can do for a limited budget of x?*. Besides the use of multiple attributes our focus should also be on adding the possibilities of time based analytic options. This would allow for representing more realistic attack scenarios, e.g. *what is the possibility of cracking a password within one week?*. This time based approach will also bring with itself a need for sequential attack steps, e.g. *Getting through a firewall before searching for the secure files*. For this, literature provides the Sequential AND (*SAND*) and the Sequential OR gate (*SOR*). Therefore the SAND gate was included in our final solution. With the combination of these features it is our firm understanding that the resulting Uppaal model has analytic possibilities that greatly exceed the possibilities of *ADTool*.

Therefore the central research question in this chapter will be as follows:

Provide a AT model which can handle multiple attributes, combined with the timed domain

- *Which features are required in the AT model?*
- *How can multiple domains of attributes be implemented?*
- *What is the best option for implementing the element of time?*

2.2 State of Art in the field of Attack Trees

There are many ways to represent attack / defence scenarios. For a representation to be both useful and usable a balance between readability and formal representation is very important. Lean too much to either side of this scale and the representation either becomes very easy to read, but impossible to use in a mathematical context or very formal and impossible to read without extensive prior knowledge. In this work we focus on avoiding the second scenario because of the naturally high involvement of business lines in every security related design decision. Due to the possible lacking technical knowledge on the subject a readable scenario will more likely be accepted and implemented.

The basis of this chapter is a paper by Kordy et al. [32] which provides a state of the art on *attack and defence modelling approaches*. In this paper the origin of graphical attack modelling is defined as threat logic trees introduced by Weiss [45]. From this original concept more than 30 different methods of modelling attack scenarios have grown. Most of them can be seen as an extension of the original model in one or more dimensions, which includes defensive components, timed actions, ordered actions, dynamic aspects and different types of quantifications. The most important of these extensions are discussed in this section. Kordy et al. [32] generally focusses on *directed acyclic graphs* (DAGs) and divides all the methods into two large subcategories, those based on threat trees and those based on Bayesian networks.

Bayesian networks are a graphical modelling formalism which focusses on random variables and the conditional dependencies between nodes. Since the main subject of this work does not involve this method, we will only quickly introduce these networks.

A great advantage of using a tree based representation over Bayesian networks is that most of the analysis algorithms are linear with respect to the number of nodes. It has to be noted that, even though with Bayesian networks a lot of algorithms are infeasible in worst case, in normal cases this effect will be limited due to the underlying cycle-free structure [32].

First we will introduce Simple Attack Trees and secondly the *ADTree* formalism and finally give an overview of the most relevant extensions provided in the review paper [32]

Simple Attack Trees

Attack Trees first were introduced by Schneier in a paper in 1999 [9]. The proposed concept is a tree structure with the attackers goal as its root and the basic attacks in the leaves. All the intermediate nodes are refinements, either an AND or an OR gate, which are respectively triggering if all or any of the children are triggered.

Attack-Defense Trees

ADTrees are introduced by Kordy et al. [29] and focuses on the gap left in the formalism presented by Schneier [9]. Simple Attack Trees only focus on the attackers perspective and leave gaps in the analysis of which defensive moves will be most effective, *ADTrees* provides the defensive side with countermeasure options, e.g. *applying extra security layers* or *increasing patrols.* The model does this by introducing defensive nodes as an inverse of the already existing nodes, this means the defensive nodes can also be *disjunctive, conjunctive or basic actions.* To maintain simplicity the terminology of *ADTree* is limited by only having a single node of the opposite type as its child. Thus, a node can have multiple children of the same type (*refinements*) and zero or one child of the opposite type (*countermeasure*). Note that this also allows for the possibility in which a node can have a countermeasure followed by another attack followed by another countermeasure.

Generally the analysis of the *ADTree* can be seen as a game between two players, one (attacker) attempting to get into the root node as easily as possible and one (defender) trying to stop this from happening. What is left can be seen as a turn-based game, i.e. in a situation were a communication channel exists between *Alice* and *Bob* and an unwanted listener (*Eve*) this game can have the following 'moves'. *Eve* installs a keylogger, which results in *Bob* introducing a second authentication factor. As a result *Eve* installs malware and so on.

As an example let us introduce *Entering a Room* (Figure 2.1) which models a possible attack on a server room, with the end goal of entering the server room. There are three disjunctive ways of achieving this goal, through the *Window*, *Door* or *Wall.* Before one can use the window to enter the room, the attacker must *Climb* up to the window and *Break the Glass*, in a non-sequential conjunctive approach since both the actions must be done in a non-specific order. If the Attacker decides to use the door, first he/she has to use the *Key* followed by opening the door, this is a sequential conjunctive choice. Finally the attacker could resort to the more destructive option of entering the room. To go through the wall the attacker can either use a form of explosives or a hammer. Either one of these will work, hence this is a disjunctive choice.

Furthermore, the leaves have certain attributes. For understandability we have greatly limited the amount, but in a normal situation there will be many more attributes. In this example, we have time (i.e. the event occurs between x and y minutes) and *boolean* *variables*, which indicate if a certain tool is required to be available. Assuming that an potential attacker has explosives and a maximum available time of 5 minutes, they can potentially use the *Explosive* to get through the *Wall* and *Enter the Room* that way.

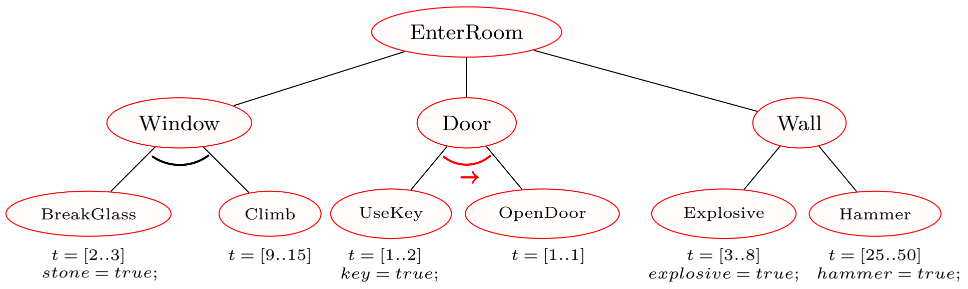

EnterRoom
Window
Door
Wall
BreakGlass
Climb
UseKey
OpenDoor
Explosive
Hammer
$t = [2..3]$ $stone = true;$
$t = [9..15]$
$t = [1..2]$ $key = true;$
$t = [1..1]$
$t = [3..8]$ $explosive = true;$
$t = [25..50]$ $hammer = true;$

Figure 2.1: Example 1: EnterRoom (time based)

ADTree comes with its own formal representation [29], which is named ADTerms and allows to express the AT in a simple yet formal way. Replacing the sequential AND with a normal AND in Figure 2.1 would result in the following ADTerms representation:

$$\vee^p(\wedge^p[BreakGlass, Climb], \wedge^p[UseKey, OpenDoor], \vee^p[Explosive, Hammer])$$

The following section will address the most relevant extensions.

OWA Trees

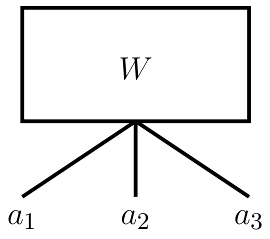


Figure 2.2: Example of an OWA-gate

A simple extension of attack trees was proposed by Yager [47], and replaces the AND and OR nodes with *ordered weighted averaging* (OWA) nodes. Where AND and OR nodes are very restrictive (either all or at least one) the OWA operators allow quantifiers such as *most*, *half*, *some*, etc. These trees are most useful when the model entails probability. Figure 2.2 shows an example of such an *owa*-gate. In this figure the weight factor (W) is the threshold, e.g. 80% and the attributes (a1 .. a3) store values which contribute to the threshold. Once enough children have been triggered the whole sub-tree will be triggered onwards.

Enhanced Attack Trees

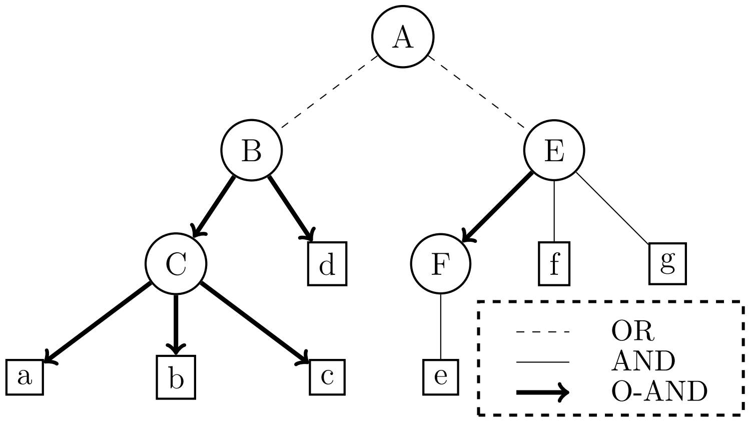


Figure 2.3: Example of an Enhanced Attack Tree

Enhanced ATs have been introduced by Camtepe and Yener [16] to support an intrusion detection engine with timed events. Such an engine attempts to detect intrusions on a system or environment. Detecting intrusions in retrospect of ones that are currently happening, which allows possible counteractions to be undertaken, before any further damage has been done. One of the most relevant features is the possibility of modelling sequential behaviour, which enforces certain basic elements to be executed in order. In Figure 2.2 you can find an Enhanced Attack Tree, it contains OR gates, AND gates and the new O-AND. This Ordered AND enforces the sequential behaviours described earlier. In other work similar approaches are Sequential AND and Priority AND. In essence these gates are similar, however all have slightly different definitions regarding ordered start times, ordered finish times etc.

Improved Attack Trees

Improved Attack Trees aim at dealing with security risks in a space-based information system. Proposed by Wen-ping and Wei-min [35] the trees model attacks on e.g. information links. Improved Attack Trees supports the OR / AND / SAND functionality. The *Sequential AND* ensures the order of its children and closely resembled the sequential behaviour we want to model in our implementation and will therefore be imported in our Uppaal model.

Defense Trees

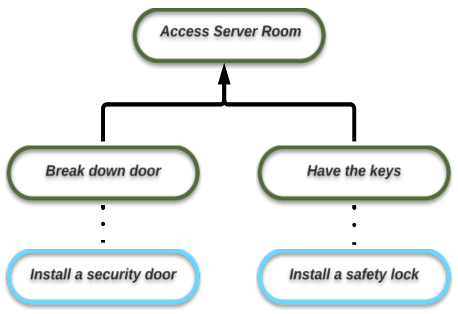


Figure 2.4: Example of a Defense Tree

In 2006 Bistarelli [13] presented defense trees as an extension to provide a possibility to model countermeasures within the leaf nodes of the tree. An example can be found in Figure 2.4, the attack *Break down door* has a countermeasure *Install a security door* and the 'attack' *Have the keys* has *Install a safety lock* as its counter. The analysis methods proposed have a formal basis and are successfully applied into the fields of economics and game theory. Two extensions of these trees are *Attack Countermeasure Trees* and *Attack-Defence Trees*

Attack Countermeasure Trees

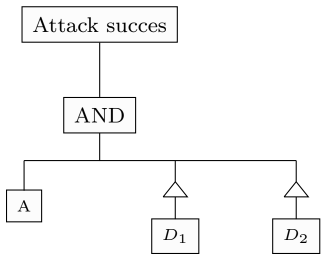


Figure 2.5: Example of an Attack Countermeasure Tree

Proposed by Roy et al. [39][40] these *Attack Countermeasure Trees* are closely related to defense trees, but with the added benefit of allowing countermeasures to be attached to intermediate nodes, allowing for more modelling possibilities. The difference being that a countermeasure is attached directly onto a sub-tree, allowing it to change the outcome of a whole sub-tree. This tree also allows for quantitative analysis as it extended by adding probability and detection nodes to further specify steps in an attack. An example of an Attack Countermeasure Tree can be found in Figure 2.5

and originates from Roy [40]. In this example A is a regular attack step (*Installing a keylogger*, whereas D_1 and D_2 are detection events, in this specific case that could be a *keystroke* or a *mouse movement*. The attack will be a success is the keylogger is installed and one of the key events is triggered. Finally this formalism introduces and uses the *k-out-of-n* gate, which triggers if a certain amount of children have been triggered. This gate is very interesting for further extensions on the Uppaal model created in this chapter as it behaves rather similarly to our AND gates, but with a lower threshold.

Protection Trees

Protection trees is a formalism that is specified around the idea that every company or organisation has a limited amount of resources, which should be used most efficiently. Protection Trees allows the users to allocate resources towards certain defensive possibilities and uses this to provide the optimal way to spend these resources. This methodology introduced by Edge et al. [19][20] and is very similar to ATs, but most nodes represent defensive moves. Such a tree is generated out of the relevant AT and inverted by finding defensive measures for every leaf. Furthermore it is enriched with three metrics, i.e. probability of success, financial costs and performance costs. These metrics allow for a way to calculate the optimal spending of these limited budgets. A noteworthy fact is that this formalism has been used to distribute budgets for US Department of Homeland Security in 2006 [19].

Countermeasure graphs

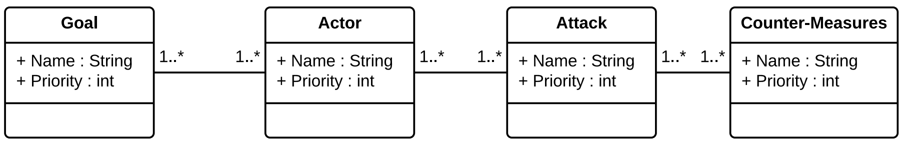


Figure 2.6: Meta-model used for Countermeasure Graphs

Countermeasure Graphs flow from a concept introduced by Baca and Peterson [10] in 2010 and prioritises a countermeasure based on the internal links it has with possible *attacks*, *countermeasures* and *goals*. Using the meta-model in Figure 2.6, one *Goal* has 1-to-many *Actors*, which in turn have multiple possible *Attacks*. And finally these attacks have multiple *Countermeasures*. This generation process blows up when there are many goals/actors or attacks and creates a vast information web. Analysis methods will deduce the most useful countermeasure for a certain situation. Baca and Peterson [10] do this by

- prioritizing **Goals** by their destructive result (either cost or stability)
- prioritizing **Agents** by their threat to the attached goals.
- prioritizing **Attack** by their chance of success.
- prioritizing **Countermeasures** by their efficiency in preventing the connected attacks.

This prioritisation process is done by assigning values to every item and simply counting the result, this system not very complicated and greatly reduces the amount of information to be further processed.

Extended Fault Trees

Extended Fault Trees, presented by Igor Nai Fovino[21] combine the failures of a system, as already represented by fault trees, with the deliberate abuse of these failures by malicious users. In essence these trees are similar to normal fault trees, only the basic elements have been altered to cover both random failures as deliberate attacks. This formalism has been included because it provides an easy extension into expanding the *Basic Elements*. Adding random failures to work in collaboration with deliberate attacks can realistically model a more opportunistic attacker. For example if an attacker sees a broken alarm he or she might be persuaded to commit to the crime.

Insecurity Flows

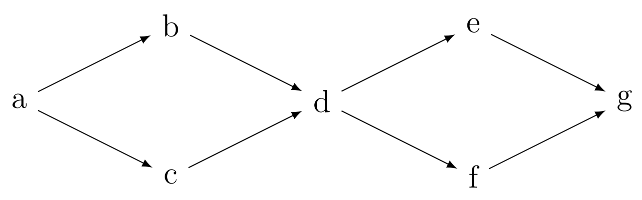

$$\begin{tikzcd}
 & b \arrow[dr] & & e \arrow[dr] & \\
a \arrow[ur] \arrow[dr] & & d \arrow[ur] \arrow[dr] & & g \\
 & c \arrow[ur] & & f \arrow[ur] &
\end{tikzcd}$$

Figure 2.7: Example of a simple Insecurity Flow

Insecurity Flows is a model from Moskowitz and Kang [36] to describe risk assessments. It combines graph theory and discrete probability to analyse possible ways how invaders can penetrate through security weaknesses. This model is very similar to reliability block diagrams used in reliability engineering [44]. An example in Figure 2.7 uses a source (attacker, node *a*) and one or multiple sinks (security breaches, node *g*) to provide a graphical representation of a security assessment. Based on the complete system, the probability that the insecurity flow can pass through the modelled security can be calculated. A major difference between this method and ATs is that in case of multiple attackers goals, it can calculate the result of multiple attack routes in one go. Regarding the analysis part, since flow networks are a well explored field in mathematics a variety of methods for optimization and analysis are already present.

Formalism	Extra gates	Tool Support	Formal
OWA trees	OWA	No	Yes
Enhanced Attack Trees	O-AND	No	Yes
Improved Attack Trees	SAND	No	No
Defense Trees	defensive	No	Yes
Attack-Defense Trees	SAND, defensive	Yes	Yes
Attack Countermeasure Trees	defensive, k-out-of-n	no	no
Protection Trees	inverse	No	No
Countermeasure Graphs	-	No	
Extended Fault Trees		No	Yes
Insecurity Flows		No	Yes

Table 2.1: Overview of the Attack Tree Formalisms

Selection of Attack Tree features

Based on our requirements and the available research presented in the previous section we made the following design choices for our chosen Attack Tree.

The **basis** will be formed by the trees used in ***ADTool*** from Kordy et al. [31] without the defensive part. This formal basis already has the **AND** and **OR** gates and to extend the analysis to sequential behaviour we have also included the Sequential AND gate. Even though the version of *ADTree* that is used in *ADTool* does not have the **SAND** gate. However *ADTools* underlying structure, *ADTree*, already has an sequential extension in another paper from Kordy [25] and formalisms like *Enhanced Attack Trees* [16], *Improved Attack Trees* [35] have already fully adapted this sequential behaviour.

Since the basis is formed by the Attack Trees used in *ADTool* the choice for ***ADTool*** as the **input tool** of choice is an obvious one.

Bigger ATs often have identical sub-trees, often parts which are used in a multitude of attacks. During analysis, once a sub-tree has been calculated every other encounter will result in duplicate work. To avoid these duplicates our implementation allows for the **sharing of sub-trees**.

Besides the attack tree itself another factor has to be considered when realistically representing threat scenarios, i.e. the attacker itself. Depending on how certain factors the whole focus might change. If an attacker is for example skilled with a computer he or she will most likely prefer a different attack than an attacker who prefers a more physical attack. This is what is called an *Attacker Profile* and which factors are most import will be explored in the next section.

2.2.1 Attacker Profiles

For our Attack Trees to have more analytical value it is important to have proper parameters in the leaves. To fully determine relevant parameters and how these relate to the real world, we will look into Criminology and Crime Science. *Criminology* is a sub-field of sociology, which draws on other disciplines to predict and prevent criminal behaviour focused on what drives / motivates the criminal. *Crime Science* also prevents crimes, but focusses on the crime itself. In this discipline every person is viewed as a potential criminal if the circumstances allow it. Where as in Criminology the criminal is someone with a tendency to commit criminal activities, either due to his upbringing or current living situation.

Money or profit is generally accepted to be the primary motivator for the gross of cyber crimes, with a few exceptions. These exceptions are often motivated by presumed moral superiority of the attackers. A few examples for the offenders in this category are Anonymous, Lulzsec or similar groups classified as *Hacktivists* (a portmanteau from Hackers and Activists). Attacks from these groups are certainly happening and should not be neglected, however the main focus of organized cyber crimes remains making a profit. According to Kshetri in 2006 [33] the simple economics of cyber crimes boils down to the following formula: $M_b + P_b > O_p + O_{cm} P_a P_c$ where the parameters are defined as follows:

- M_b: **Monetary Benefits**. How much money is the cyber crime yielding to the criminal.
- P_b: **Psychological Benefits**. More than with regular criminal activities early cyber crime activities were often not motivated by financial gain, but instead by fun or challenge. Today this is still a partial motivator.
- O_p: **Psychological Costs**. Or named negative psychological effects. Do hackers feel guilt or remorse for their crimes? Most experts agree that the barrier of

the internet greatly reduces overall guilt, however there is some sort of ethics in thievery.

- O_{cm}:**Monetary Opportunity Costs**. An estimate on how expensive the penalty will be, this might be a simple fine of \$150, but can also be a numeric value for a year in prison, Kshetri [33] uses a normal year salary of \$ 20.000, thus a three year sentence would cost the attacker \$ 60.000.
- P_a: **Probability of an arrest**, chances of arrest are really low for cyber crimes, Kshetri [33] even suggests that smart cyber criminals which are operating over country borders are nearly untouchable.
- P_c: **Probability of a conviction**. Even if the criminals are arrested, international laws are not identical and arguing new crimes in court is not easy. e.g. cyber theft is different from regular theft, as is it often considered to be closer to copying than to stealing.

Insider crimes are also a fairly well investigated area, primarily because in these cases both the criminal and the victim can be investigated. Also considering the damage that these types of crimes inflict, i.e. simple identity theft might empty out a single account whereas a bank employee can do this for a multitude of accounts. Dhillon investigated two cases in 2001 [18], *the Kidder Peabody case* and *the Daiwa Bank Scandal*. Here the damages were \$ 339 million and \$ 1.1 billion respectively. These two cases were both possible due to the absence of basic safeguards (monitoring, monthly asset reviews etc). These safeguards give an organisation a better position in these type of events. In both cases, the insider criminals got too much unsupervised access and this resulted in the opportunity to commit (small) illegal acts. Once the smaller things went unnoticed, those smaller things slowly escalated into the bigger numbers until it eventually became too big to go unnoticed.

Futher studies performed by an research group from the university of California and Google [15], suggest a great level of professionalism within the network of cyber criminals. Within their data they found that during non-automated attacks the attackers took their time to profile the account they had gained access to. Based it on roughly 3 minutes of analysis, the attacker followed up by one of the predefined base strategies, e.g. empty the bank account, using the clean account status for sending spam or leave the account as is, but installing monitoring filters. The researchers noticed the wave of attacks starting everyday around the same day and even detected a synchronized pause halfway, suggesting something like office hours with a lunch-break. An noteworthy paper by Cormac Herley [23] tries to explain why the so called "Nigerian scammers" still send really obvious emails telling them about the famed African Riches which can be yours for only $1000. His conclusion comes down to victim selection, attackers have a limited amount of time to maximize their profit and sending out really obvious emails will only attract the most gullible of victims, which will increase their success-rate and thus profit.

2.3 Attack Tree Domains

In providing a formal definition for the Attack Trees used within this thesis, we will closely follow the formal definition provided by Schneier in [9] and the formalization provided by Jhawar et al. [25]. Below an Attack Tree grammar is provided together with its translation into Series-Parallel Graphs is provided.

2.3.1 Attack Trees

Let $\mathbb{B}$ denote a set of all possible basic attack steps an attacker can take. Simple attack trees are formalized in [25] based on the trees introduced by Schneier in [9]. These attack trees are closed terms over the signature $\mathbb{B} \bigcup \{OR, AND\}$, generated by the following grammar, where b $\in \mathbb{B}$ is a terminal symbol.

$$t ::= b \mid OR(t, ..., t) \mid AND(t, ..., t).$$

Jhawar et al. [25] also added the Sequential Conjuction, which they named a SAND gate. The signature changed to $\mathbb{B} \bigcup \{OR, AND, SAND\}$ and this also extended the grammar into the following.

$$t ::= b \mid OR(t, ..., t) \mid AND(t, ..., t) \mid SAND(t, ..., t).$$

The universe of SAND attack trees is denoted by $\mathbb{T}_{SAND}$. All attack trees in the rest of this thesis will originate from this universe.

When considering attributes in attack trees they should be calculated in intermediate nodes, for example consider an AND gate with two children with a *cost* attribute, the intermediate node should know how to be calculated based on the children. However when considering multiple different attributes there is need for some separation, i.e. *time* should not be added to *damage*. These two attributes should be considered separate, in their own *Attribute Domain*.

2.3.2 Attribute Domain

Again following [25], we define an attribute domain for every attribute (A_α) as a tuple $D_\alpha = (\ V_\alpha,\ \triangledown_\alpha\ ,\ \triangle_\alpha\ ,\ \lozenge_\alpha\ )$, where V_α is a set of values and $\triangledown_\alpha$, $\triangle_\alpha$, $\lozenge_\alpha$ are families of k-ary functions (one family for each type of refinement node) of the form $V_\alpha \times \cdots \times V_\alpha \to V_\alpha$ associated to respectively OR, AND and SAND refinements. This provides a formula for every domain / refinement combination, which can be used to calculate attributes bottom up, eventually ending up in the root node. As an example a *cost* attribute needs to be added in an AND refinement, this gives us the formula $\triangle_\alpha(x_1, \cdots, x_k) = \sum_{i=1}^{k} x_i$ for this domain.

Every attribute for these attack trees is a pair $A_\alpha = (\ D_\alpha,\ \beta_\alpha\ )$ formed by an attribute domain D_α and a function $\beta_\alpha : \mathbb{B} \to V_\alpha$. This is called a basic assignment for A_α. This creates a link between an attribute and a domain. At a single refinement multiple (different) formulas can be used when the attack has multiple attribute domains.

2.3.3 Domains in ADTool

ADTool provides a basic set of attribute domains: *time*, *cost*, *satisfiability* and *reachability*, all of which have a minimal and a maximal interpretation. The following will list the domains and how they respectively handle the OR, AND and Sequential AND refinement (SAND)

Difficulty

Difficulty takes a minimum of the difficulty when it is a disjunctive refinement and will take the maximum when it encounters a (sequential) conjunctive refinement.

$$\begin{aligned}
\triangledown_\alpha(x_1, \cdots, x_k) &= min(x_1, \cdots, x_k) \\
\triangle_\alpha(x_1, \cdots, x_k) &= max(x_1, \cdots, x_k) \\
\lozenge_\alpha(x_1, \cdots, x_k) &= max(x_1, \cdots, x_k)
\end{aligned}$$

Time

Time is calculated by its minimum in disjunctive refinement, maximum in its conjunctive refinement and for the sequential AND it will add the times of its children.

$$\begin{aligned}
\triangledown_\alpha(x_1, \cdots, x_k) &= min(x_1, \cdots, x_k) \\
\triangle_\alpha(x_1, \cdots, x_k) &= max(x_1, \cdots, x_k)
\end{aligned}$$

$$\Diamond_\alpha(x_1,\cdots,x_k) = \sum_{i=1}^{k} x_i$$

Cost

Cost takes a minimum difficulty of its children when it is a disjunctive refinement and will take the maximum when it encounters a (sequential) conjunctive refinement

$$\begin{aligned}\nabla_\alpha(x_1,\cdots,x_k) &= min(x_1,\cdots,x_k)\\ \triangle_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\\ \Diamond_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\end{aligned}$$

Maximal consumption

Maximum consumption is calculated by maximizing over an disjunctive refinement and adding all the consumption in the case of a (sequential) conjunctive gate.

$$\begin{aligned}\nabla_\alpha(x_1,\cdots,x_k) &= max(x_1,\cdots,x_k)\\ \triangle_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\\ \Diamond_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\end{aligned}$$

Reachability (parallel)

Reachability in parallel minimizes over OR gates, and maximizes over (Sequential) AND gates.

$$\begin{aligned}\nabla_\alpha(x_1,\cdots,x_k) &= min(x_1,\cdots,x_k)\\ \triangle_\alpha(x_1,\cdots,x_k) &= max(x_1,\cdots,x_k)\\ \Diamond_\alpha(x_1,\cdots,x_k) &= max(x_1,\cdots,x_k)\end{aligned}$$

Reachability (sequential)

Reachability in series still minimizes over the OR gates, but takes the summation of its children on (Sequential) AND gates.

$$\begin{aligned}\nabla_\alpha(x_1,\cdots,x_k) &= min(x_1,\cdots,x_k)\\ \triangle_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\\ \Diamond_\alpha(x_1,\cdots,x_k) &= \sum_{i=1}^{k} x_i\end{aligned}$$

Satisfiability

Satisfiability is the passing along of booleans, therefor will use the $\vee$ when it encounters an OR and will $\wedge$ all its children on a (sequential) conjunctive refinement.

$$\begin{aligned}
\triangledown_\alpha(x_1,\cdots,x_k) &= x_1 \vee \cdots \vee x_k \\
\triangle_\alpha(x_1,\cdots,x_k) &= x_1 \wedge \cdots \wedge x_k \\
\Diamond_\alpha(x_1,\cdots,x_k) &= x_1 \wedge \cdots \wedge x_k
\end{aligned}$$

How to handle domains

Even though the next Chapter will focus on how the whole AT transforms into a Uppaal model, this part will quickly address how domains are handled. For example take the *Cost* domain. If a *Cost* domain is present the Uppaal model will have an *handleCost(int id)* in its declarations together with an int result_Cost variable that will store the global Cost value for the current trace. For every leaf that gets activated this function is called with its respective *id*. For cost the functionality is simple, it will add the cost of this specific leaf to the global value. The code snippet responsible for the process described here can be seen below.

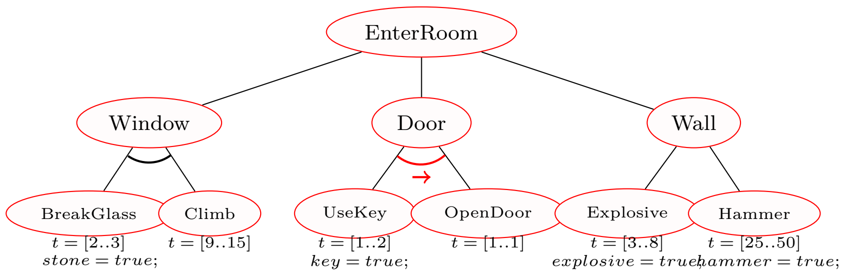
```
int result_Cost = 0;

int Cost[3] = {10, 36, 22};

void handleCost(int id) {
        result_Cost = result_Cost + Cost[id];
}
```

2.4 Examples

Our system should be able to handle models like the following examples:

Note: *the examples already use the sequential AND in addition to the normal gates.*

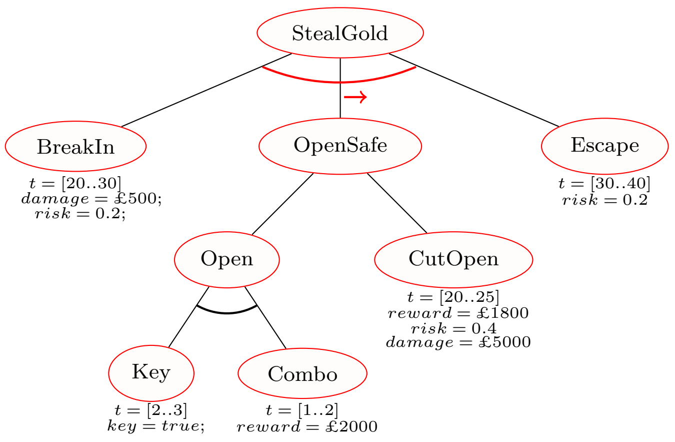
Example 1: EnterRoom (time based)



This example models a possible attack on a room, with the end goal of entering the room. There are three disjunctive ways of achieving this goal, through the *Window*, *Door* or *Wall*. Before one can use the window to enter the room, the attacker must *Climb* upto the window and *Break the Glass*, this non-sequential conjunctive approach since both the actions must be done in a non-specific order. If the Attacker decides to use the door, first he/she has to use the *Key* followed by opening the door, this is a sequential conjunctive choice. Finally the attacker could resort to the more destructive option of entering the room. To go through the wall the attack can either use a form of explosives or a hammer. Either one of these will work, hence it is a disjunctive choice.

Furthermore, the leaves have certain attributes. For understandability we have greatly limited the amount, but in a normal situation there will be many more attributes. In this example, we have time (i.e. the event occurs between x and y minutes) and *boolean variables*, which indicate if a certain tool is required to be available.

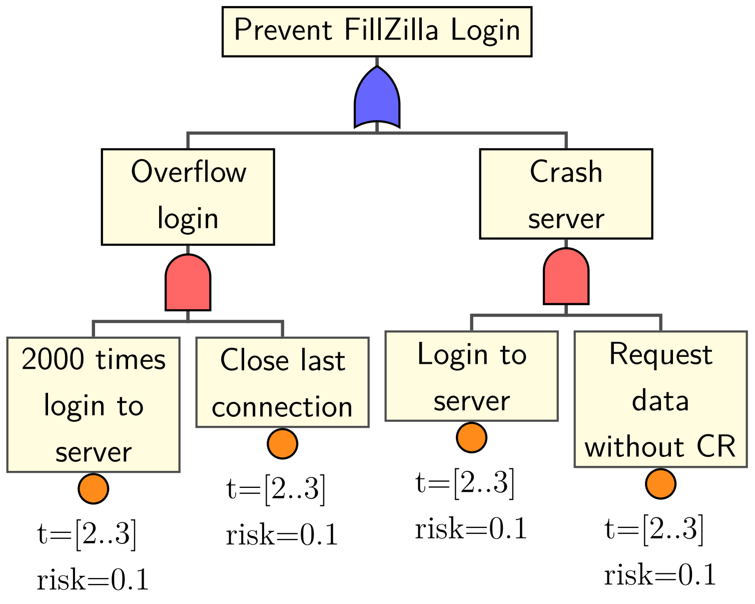
Example 2: StealGold



This example originates from Arnold et al. [8] and models a situation in which the attacker attempts to steal gold and escape. The general goal can be achieved by *Breaking in*, *Opening the safe* and *Escaping*, these actions must occur in order, which makes this a sequential conjunctive gate. The *BreakIn* and *Escape* are basic events and are not defined further. However opening the safe can be done in two ways, it can either be opened by force (*CutOpen*) or opened by having the key and knowing the combination (*Open*), this is a disjunctive choice. If the attacker wants to open the safe without using force, both the *Key* and the *Combination* have to be obtained, this is a conjunctive option.

Again every leaf has certain attributes. Note that not all attributes are present on every leaf, if an attribute is not present we, assume it is 0 (e.g. the *damage* of the *escape* leaf is 0)

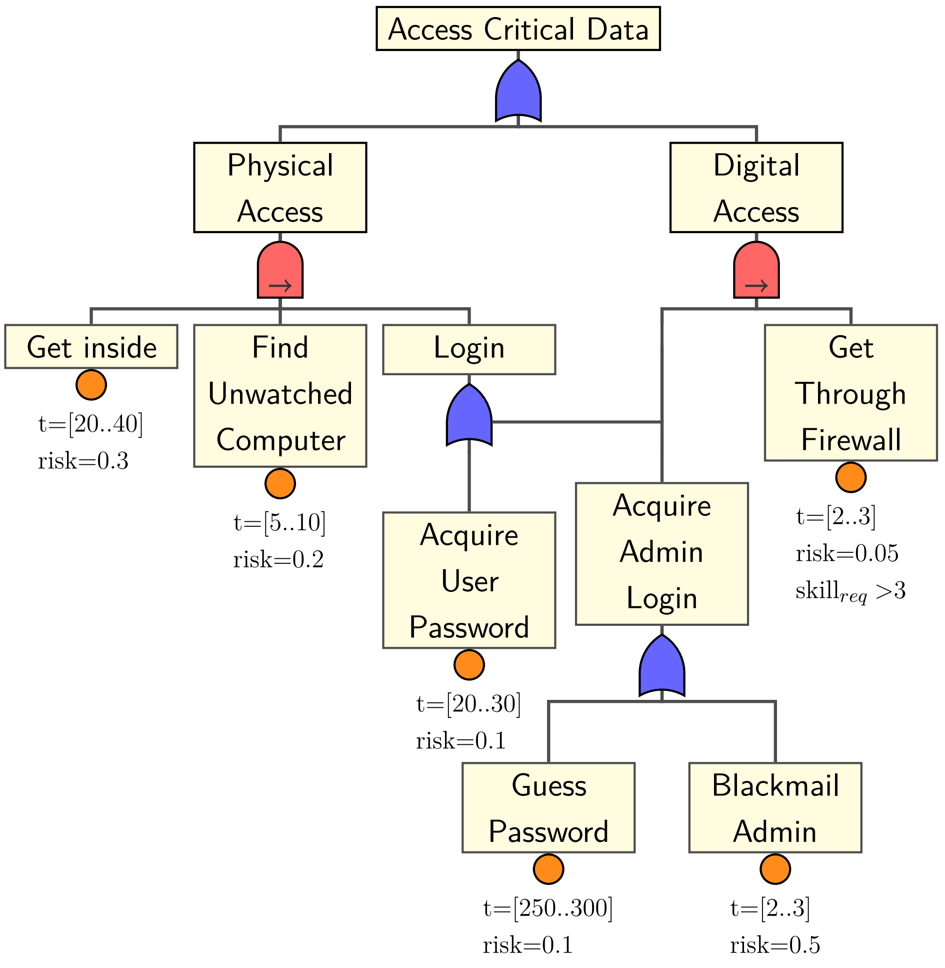
Example 3: DDOS on FileZilla



The third example models possible weaknesses in Mozilla's ftp-client (*Filezilla*) and is based on an analysis done in a paper by Sanford, Woodraska and Xu [41]. One of the general weaknesses they address is the possibility of preventing legitimate users to log-in. This can be disjunctively done either by overflowing the login or by crashing the server. In the first case the paper shows that the attacker should first attempt a minimum of 2000 logins followed by closing the last connection. In the second case (*Crashing the Server*), a legitimate login is required, if this is available this user can simply keep sending data without a CR, this will then overflow the server and make it crash.

Provided as attributes are; the time and the risk of the attack being discovered (and thus the damage can be mitigated).

Example 4: Access Critical Data (shared sub-tree)



The last example explores shared leaves and does this by using the "*Acquire Admin Login*" sub-tree in both parts of the at the root separated sub-tree. In real life the admin login credentials can often be used internally as well as externally. Apart from this, the attack tree is rather similar to previous examples. The critical data can be accessed from two possible ways, through *Physical* or *Digital* access. If the attacker chooses the physical path, the following events must occur in order: *Get Inside, Find Unwatched Computer* and *Login.* For *Digital Access* the attacker needs the admin login and the skill to get through the firewall. *Acquire Admin Login* can be done almost risk-free by guessing the credentials, but the chance of success is very limited. Or it can be done very quickly by blackmailing the admin, but this comes at a high risk.

Attributes provided are: the time and the risk of the attack being discovered (and thus the damage can be mitigated).

Graphical Representation

The first two examples are given in the style of the *ADTool* [1]. To increase the usability of our transformation we decided to use the formalism of *ADTool* as the basis for generating the required Upaal model. This choice also decided the choice of visualisation.

In this research the focus will be on a model that excels in timing issues and its readability, because this is where the essence should be, it should be readable and easy to use before anything else. Furthermore even though these choices will not make our analytical capability easier, the primary results seem promising that further analytical progress can be made by using the basis of the model. Relying on real-time automata combined with the extra features of *Uppaal SMC* [17], which allows for more statistical model checking.

Chapter 3

Attack Trees as Timed Automata

This Chapter defines the way Attack Trees (ATs) can be represented in Timed Automata. The results of Chapter 2 are used as the foundation for this Chapter. The *basic elements* in the *leaves*, the *gates* in the *intermediate nodes*, the *timed domains* and the *other types of attribute domains.*

This Chapter will focus on the following research questions.

How can Attack Trees be modelled in Timed Automata?

- *Which Timed Automata can be used for the Basic Elements and Gates?*
- *How could the implementation of multiple domains attributes be realised?*
- *How is the element of time transferred from ATs into the Uppaal model?*

3.1 Timed Automata

A timed automaton is a finite state machine which has been extended with the basic concept of time. Finite state machines use nodes to represent states that a machine can be in and use edges between these states as transitions from one state into another.

Time is included by using clocks and functions as a feature which allows for a different type of analysis. For regular Timed Automata, all the clocks progress synchronously.

To actually use Timed Automata for State Space exploration research provides several options. Some optional scientific tools are *HyTech* [22], *Kronos* [7] and *Uppaal* [11]. However since for the TREsPASS project the need arose to express ATs in a Uppaal model, this Chapter will focus on the Timed Automata used in *Uppaal.*

3.2 Uppaal

Uppaal is a tool for verification of real-time systems jointly developed by Uppsala University and Aalborg University [12]. The tool can verify systems that can be modelled as networks of timed automata extended with additional data-types, among which *channel synchronisation* is the most important. The tutorial from Behrmann et al. [12] introduces the tool with examples and covers all the basics that are required to start working with the tool.

Uppaal uses the following formal Definition 3.2.1 for Timed Automata according to Behrmann [12] and since this is the definition used for the Uppaal models, it will also the definition used in this Chapter.

Definition 3.2.1. (Timed Automaton (TA)) *A timed automaton is a tuple* (L, l_0, C, A, E, I)*, where* L *is a set of locations,* $l_0 \in L$ *is the initial location,* C *is the set of clocks,* A *is a set of actions, co-actions and the internal* τ*-action,* $E \subseteq L \times A \times B(C) \times 2^C \times L$ *is a set of edges between locations with an action, a guard and a set of clocks to be reset, and* $I : L \to B(C)$ *assigns invariants to locations.*

The basic method used to define Attack Trees as Timed Automata can be described as follows. Every node will be represented as a instance of a Uppaal Template (a process) and all these separate Timed Automata will be put in parallel to explore the state space and answer the questions fired upon the model. Question are primarily of the following nature: *can we reach state x?*, *What is the fastest way to reach Y?*, *Can we reach x, without cost exceeding z?* etc.

For example take *StealGold* AT from Example 2 (Chapter 2.4), this AT has five leaves, one OR-gate, one AND-gate and one SAND-gate. Transforming this into a Timed Automata with the required functionality has resulted in nine parallel processes, based on six different templates (*Timed Basic Element*, *Dormant Timed Basic Element*, *AND2*, *AND3*, *OR2* and an overview process *System*).

These processes communicate over broadcast channels, the sending process can use an emission on a specific channel to trigger the process on the receiving end. For there to be sufficient communication every node needs an outgoing channel to communicate upwards, for the *StealGold* example this resulted in eight broadcast channels. Every instance of a node will be listening on the channels of its children to receive their signal, i.e. *Open (AND2)* will be waiting for *Key (TimedCBE)* and *Combo (TimedCBE).*

Additionally all the dormant (*Cold*) Basic Elements wait on another signal, the left sibling of their first ancestral Sequential AND gates. i.e. *Key*, *Combo* and *CutOpen* all await activation from the communication channel of *BreakIn.*

As elements of ATs are defined as instances of Uppaal templates, the next part will display and explain the different templates that are included into our Uppaal model.

3.3 Uppaal Templates

The following set of templates made the final cut and are implemented for the Uppaal model. Every template will be accompanied with a brief description, design choices and if applicable a textual description of an alternative. If an template was not included in the final transformation process there will be a quick explanation.

Exponential Basic Element

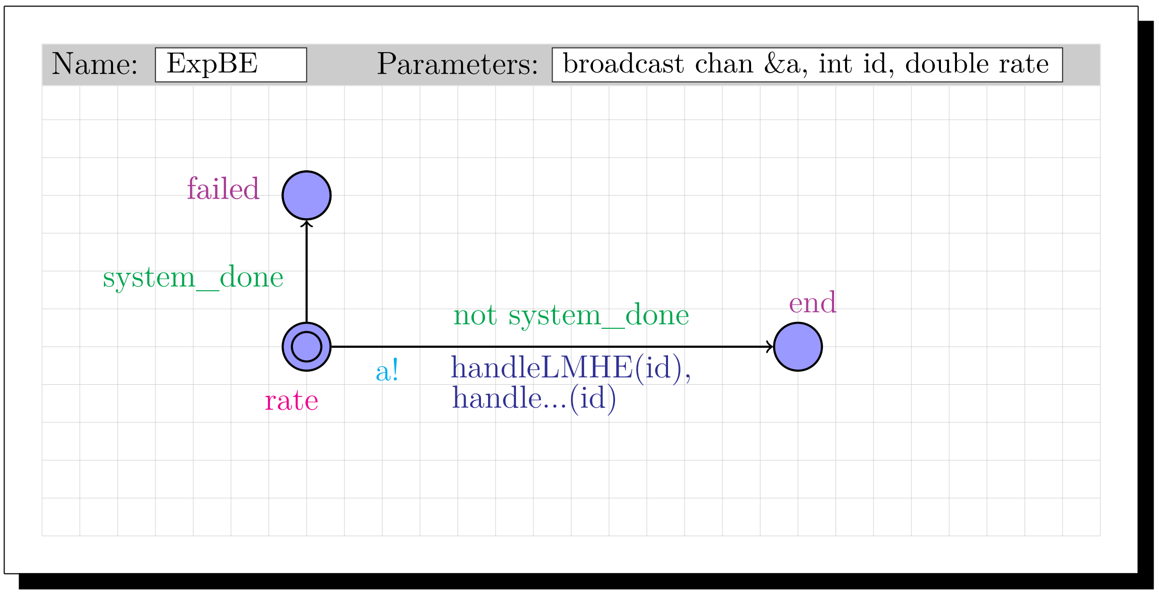


Figure 3.1: Basic Element (Trigger: Exp(1/e))

General functionality of the Exponential Basic Element

This Basic Element represents a leaf on an Attack Tree, the trigger for this type of Basic Element is an *Exponential Distribution*, hence the name *Exponential Basic Element*. Once the trigger has fired the transaction towards *end* will occur if the system is still active, this is checked based on the global boolean system_done. If the system is no longer active it will be directed to the *failed* state. During the transaction towards *end* it will both broadcast on the channel *a* (provided in the arguments) and execute the specified *handeX()* functions. The number of handle functions is dependant on the number of domains of the whole attack tree and are determined at the moment of creation. The argument that the handle function take is provided by the parameter *int id*, which is the identifier of this specific leaf.

Erlang Basic Element

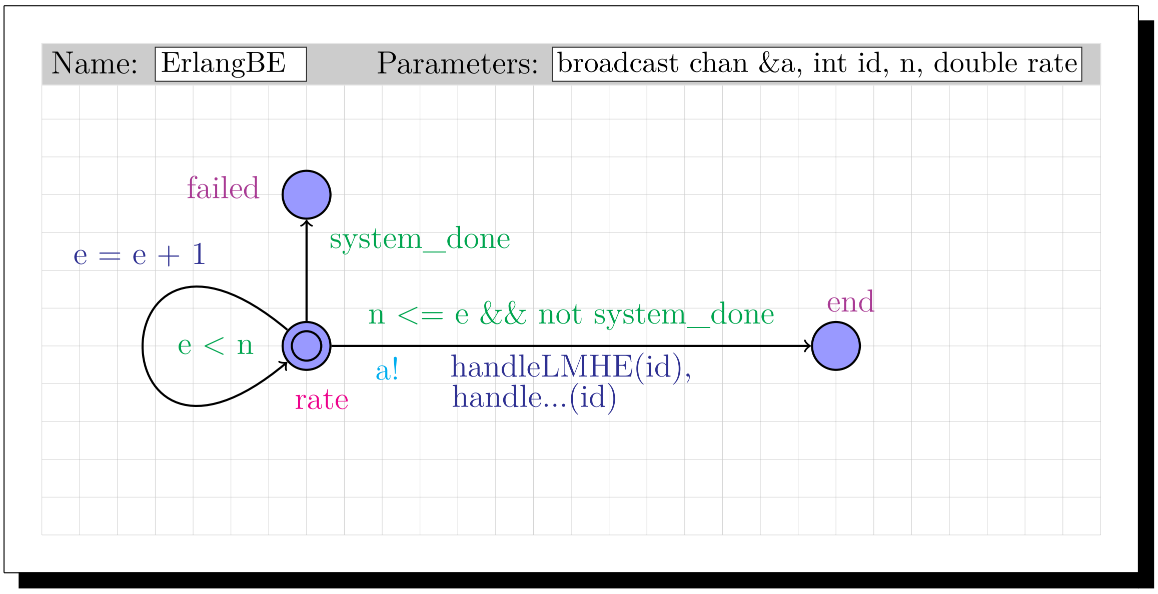


Figure 3.2: Basic Element (Trigger: Erlang(n, 1/rate))

Note that this ErlangBE is not included in the transformation. If there is a time domain available, the transformation will use TimedBE and otherwise fall back on the ExpBE. This template is included in this Chapter because of it uses in possible future work.

General functionality of the Erlang Basic Element

This Basic Element represents a leaf on an Attack Tree, these leaves can have different triggers, the trigger for this type of Basic Element is an *Erlang Distribution*. Once the trigger has fired this transaction will occur if the system is not yet done, this is checked based on the global boolean system_done. During the transaction this process will both broadcast on the channel *a* provided in the arguments and execute the specified *handeX()* functions. The number of handle functions is dependant on the number of domains of the whole attack tree and are determined at the moment of creation. The argument that the handle function take is provided by the parameter *int id*, which is the identifier of this specific leaf. An Erlang distribution originates from multiples Exponential Distributions and that is what this Timed Automata does, it keeps looping until it did its required number of exponential triggers.

Timed Basic Element

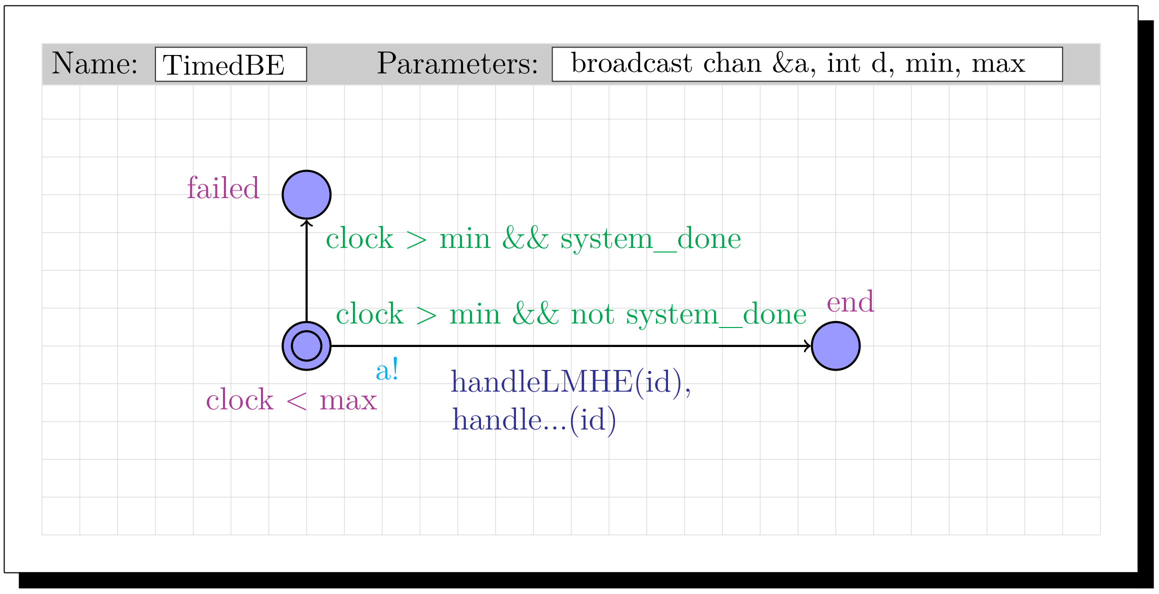

Name: TimedBE
Parameters: broadcast chan &a, int d, min, max
failed
clock > min && system_done
clock > min && not system_done
end
a!
clock < max
handleLMHE(id),
handle...(id)

Figure 3.3: Basic Element (Trigger: Time(min,max))

General functionality of the Timed Basic Element

This Basic Element represents a leaf on an Attack Tree, the trigger for this type of Basic Element is time, it will fire between two values (c and d). Once the trigger has fired this transaction will occur if the system is not yet done. During the transaction this process will both broadcast on the channel a provided in the arguments and execute the specified *handeX()* functions. The number of handle functions is dependant on the number of domains of the whole attack tree and are determined at the moment of creation. The argument that the handle function take is provided by the parameter *int b*, which is the identifier of this specific leaf. The distribution this Automata uses is a linear one between timeslot c and d.

Dormant Exponential Basic Element

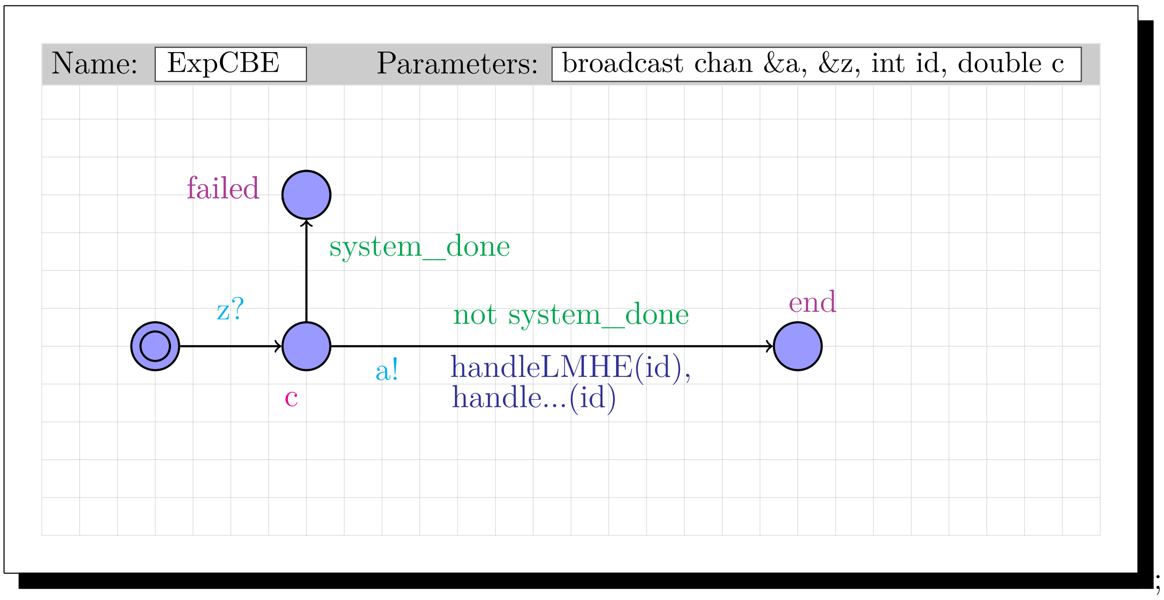


Figure 3.4: Dormant Basic Element (Trigger: Exp(1/c))

General functionality of the Exponential Basic Element

This Basic Element represents a leaf on an Attack Tree. After activation (channel z) the trigger for this type of Basic Element is an *Exponential Distribution*, hence the name *Dormant Exponential Basic Element*. Once the trigger has fired this transaction will occur if the system is still active, this is checked based on the global boolean system_done. During the transaction this process will both broadcast on the channel a provided in the arguments and execute the specified *handeX()* functions. The number of handle functions is dependant on the number of domains of the whole attack tree and are determined at the moment of creation. The argument that the handle function take is provided by the parameter *int b*, which is the identifier of this specific leaf. If a system is no longer active, but this element is activated it will end in the *failed* state.

The only difference this has to the non dormant variant is that this one first needs to be activated.

Dormant Timed Basic Element

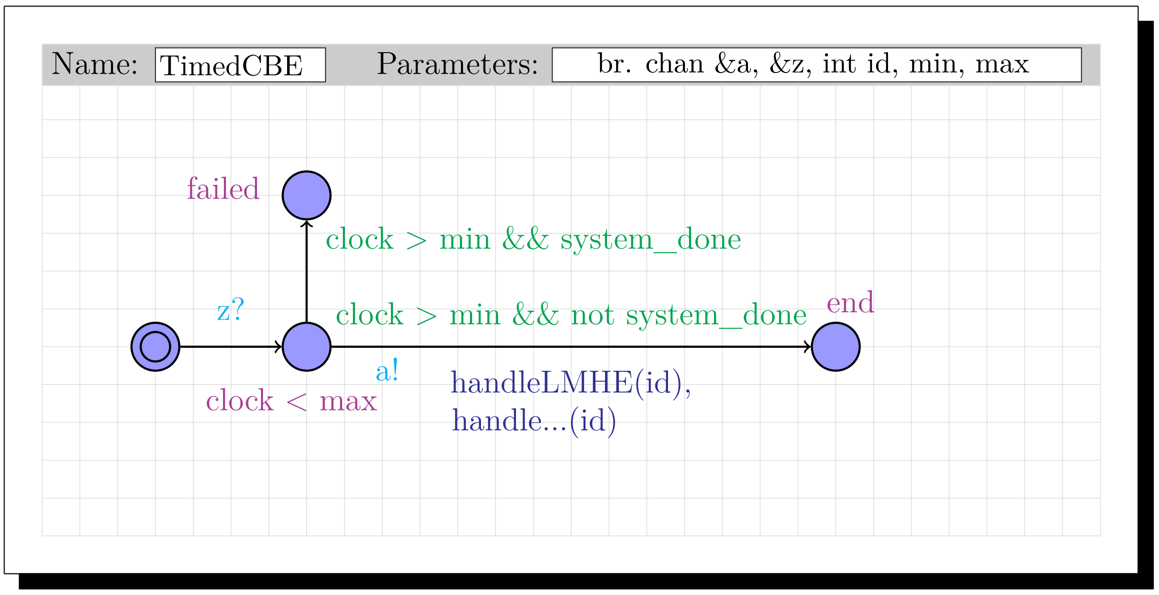

Name: TimedCBE
Parameters: br. chan &a, &z, int id, min, max
failed
clock > min && system_done
z?
clock > min && not system_done
end
a!
clock < max
handleLMHE(id),
handle...(id)

Figure 3.5: Dormant Basic Element (Trigger: Time(min,max))

General functionality of the Timed Basic Element

This Basic Element represents a leaf on an Attack Tree, the trigger for this type of Basic Element is time. This will fire between two time values (*min* and *max*). Once the trigger has fired this transaction will occur if the system is still active. During the transaction this process will both broadcast on the channel *a* and execute the specified *handeX()* functions. The number of handle functions is dependent on the number of domains of the whole attack tree and are determined at the moment of creation. The argument that the handle function take is provided by the parameter *int id*, which is the identifier of this specific leaf. The distribution this Automata uses is a linear one between time slot min and max.

The only difference this has to the non dormant variant is that this one first needs to be activated.

AND-gate, with 2 children

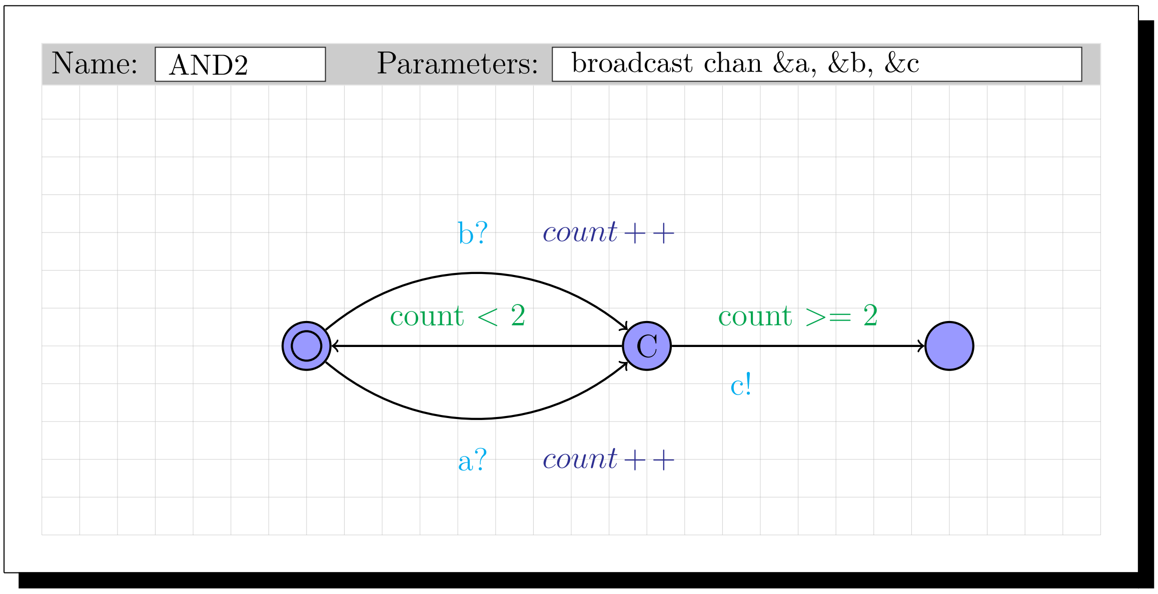


Figure 3.6: AND-gate, with 2 children

General functionality AND gate

This gate listens to two broadcast signals (a and b), which have been assigned to the broadcast signals of the children of the represented intermediate node. If both children have fired and that puts the count to 2 this template will broadcast over channel c, which has been passed along in the parameters.

Note: this template is for two children, very similar templates can be generated for more children, which will be done in the transformation if the need for this arises.

OR-gate, with 2 children

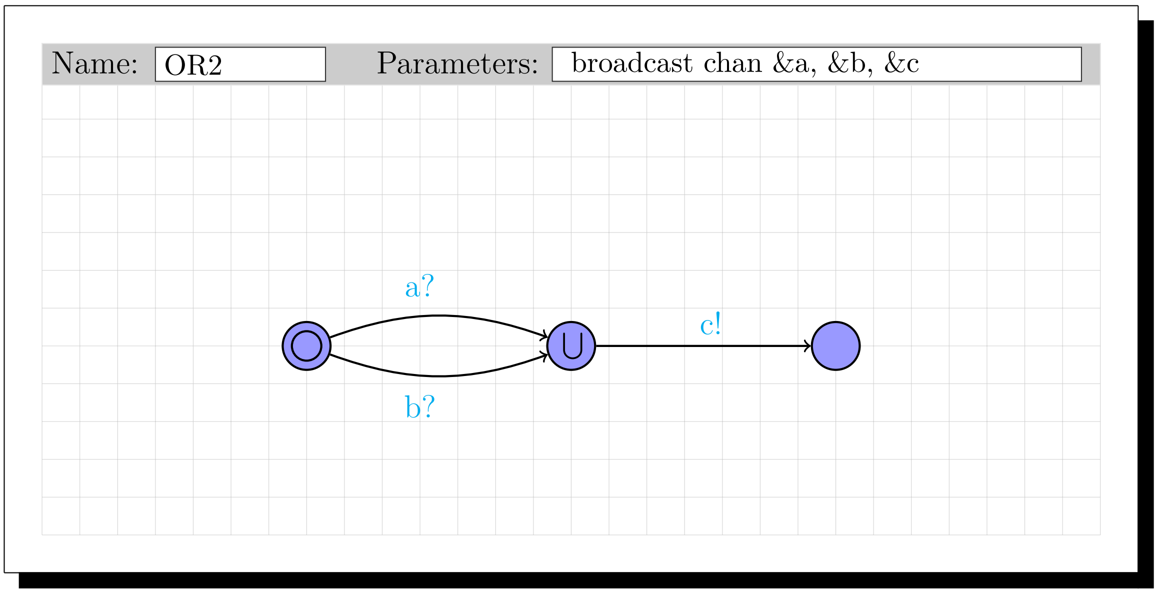


Figure 3.7: OR-gate, with 2 children

General functionality OR gate

This gate listens to two broadcast signals (a and b), which have been assigned to the broadcast signals of the children of the represented intermediate node. If one of the children has fired this template will broadcast over channel c, which has been passed along in the parameters.

Note: this template is for two children, very similar templates can be generated for more children, which will be done in the transformation if the need for this arises.

Dormant AND-gate, with 2 children

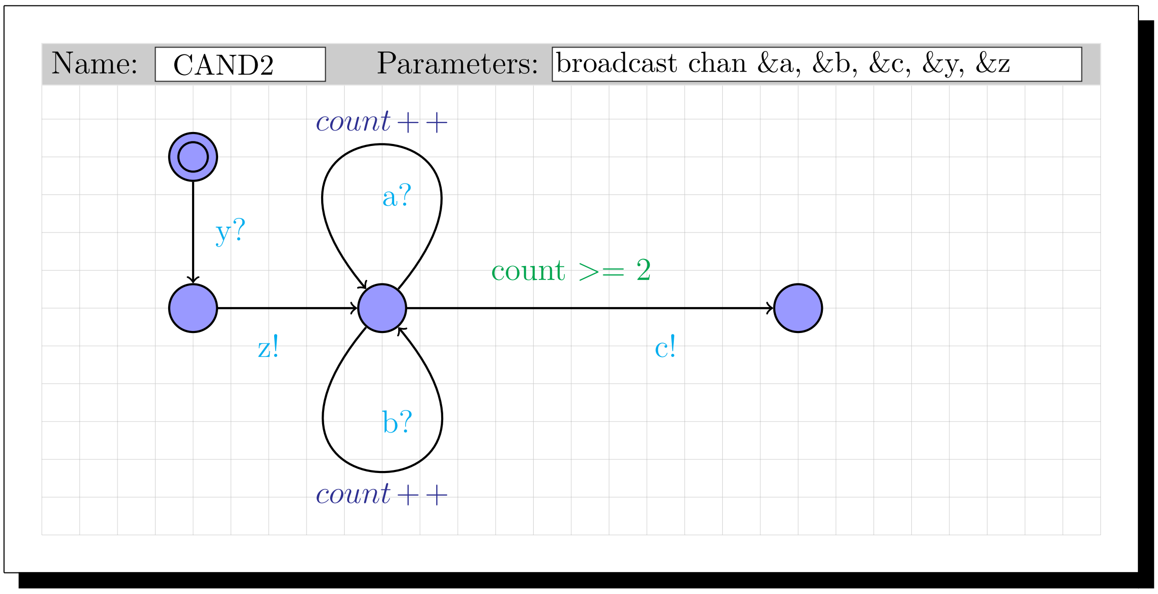


Figure 3.8: Dormant AND-gate, with 2 children

General functionality Dormant AND gate

Since this is the dormant variant it will only be used when it has a SAND gate as one of its ancestors. This gate listens to two broadcast signals (a and b), which have been assigned to the broadcast signals of the children of the represented intermediate node. If both children have fired and that puts the count to 2 this template will broadcast over channel c, which has been passed along in the parameters.

The only difference between this and its non-dormant variant is that this template is waiting for activation

Note: this template is for two children, very similar templates can be generated for more children, which will be done in the transformation if the need for this arises.

Dormant OR-gate, with 2 children

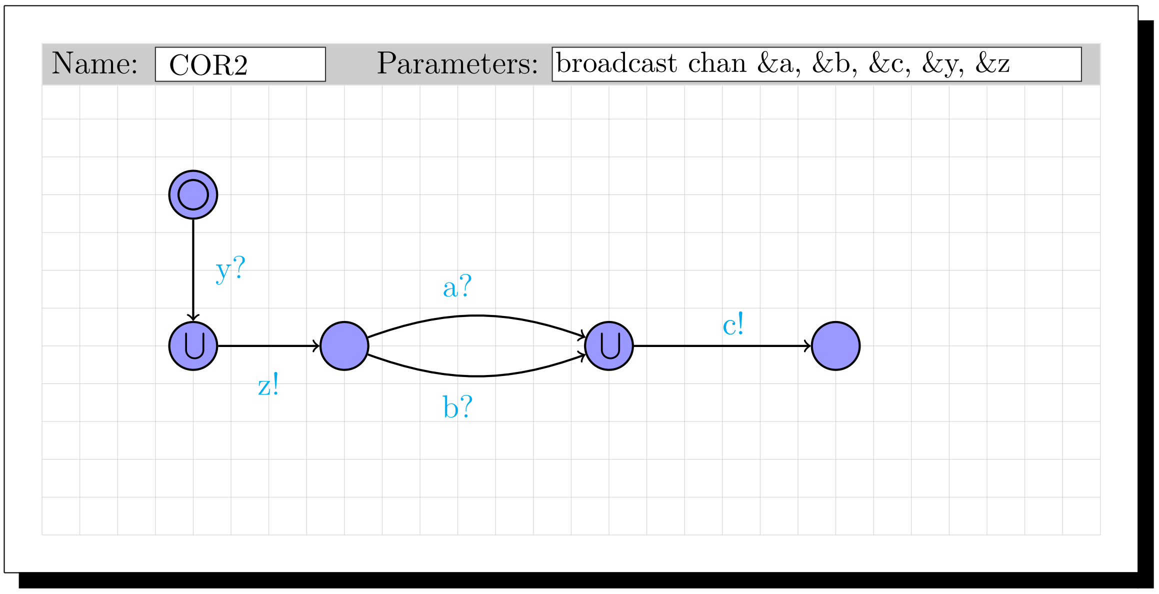


Figure 3.9: Dormant OR-gate, with 2 children

General functionality Dormant OR gate

Since this is the dormant variant it will only be used when it has a SAND gate as one of its ancestors. This gate listens to two broadcast signals (a and b), which have been assigned to the broadcast signals of the children of the represented intermediate node. If one of the children has fired this template will broadcast over channel c, which has been passed along in the parameters.

The only difference between this and its non-dormant variant is that this template is waiting for activation

Note: this template is for two children, very similar templates can be generated for more children, which will be done in the transformation if the need for this arises.

Chapter 4

Attack Tree Meta Model

4.1 Meta-Modelling in general

Model Driven Engineering (MDE) is based upon the use of abstractions to structure and clarify the intentions of the programmer rather than the specific code needed to execute this functionality on a computer. Abstractions have always been at the core of software development. Not very long ago the first abstractions made sure that the programmer could combat the increasingly complex requirements [42]. These approaches were the basis of model-driven development and helped to move the focus from programming languages onto Model Driven Engineering.

MDE also influenced the way of developing in another way: the abstraction made use of UML to abstract the design of computer programs into a model [42]. It has become common practice for most software developers to use UML models to represent their (initial) design intentions.

One step further is to represent a model as an instance of a more abstract version, this is a meta-model. Such a meta-model can be used to identify and define the construction, rules, constraints, models and theories surrounding every instance of it. An example of a framework that is used for meta-modelling is the Eclipse Modelling Framework (EMF), which is currently the industry standard.

As an example the meta-modeling will be illustrated in Figure 4.1 by showing a meta-model for a book and an instantiation of this model. Note that we made this meta-model fairly generic so that the majority of books will conform.

A *Book* is an entity with certain values, which are its *title*, *genre* and its *price*. Besides its values it also has one reference to its *Author*. This is a different entity which has values of its own, *firstname* and *lastname*. Note that the Book - Author reference is bidirectional, the *Book* has atleast one *Author*, but an *Author* can have multiple *Books*.

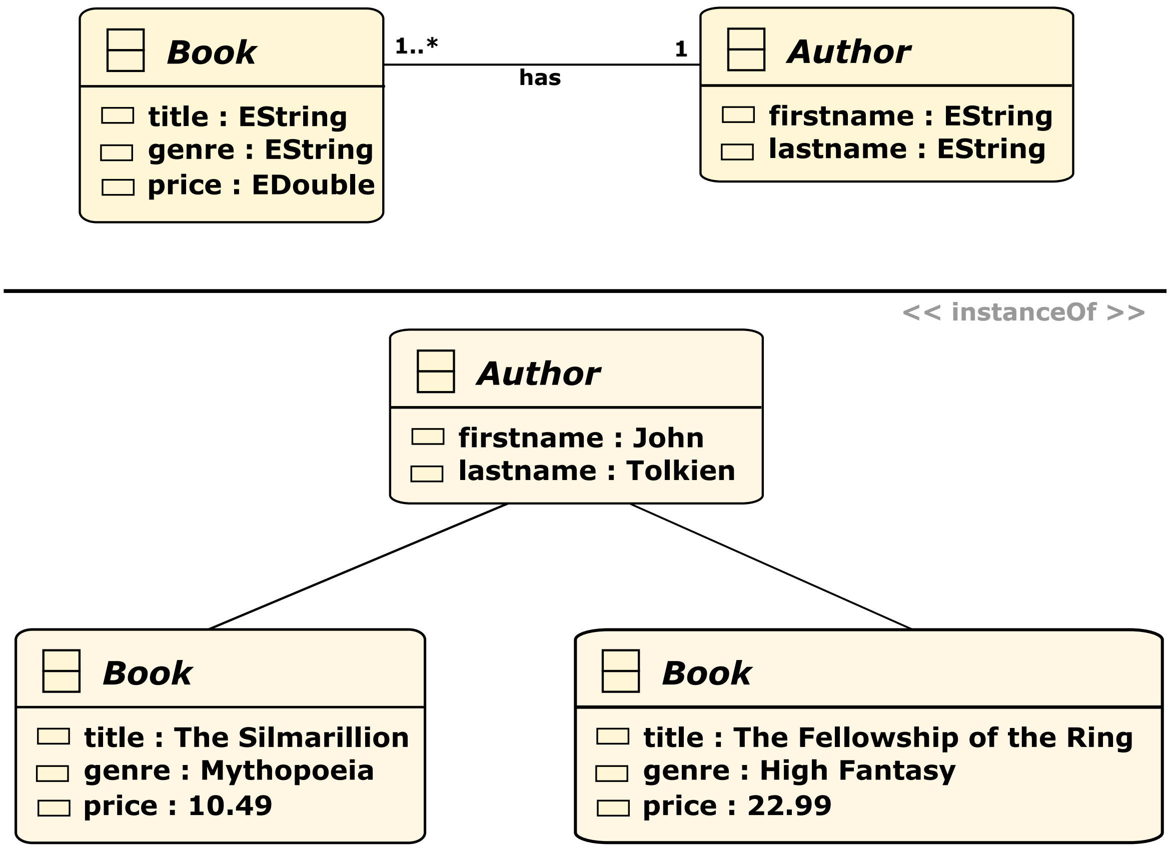


Figure 4.1: Meta-modelling example

An example of instances can be found in the bottom half of Figure 4.1 which displays two instances of this meta-model, with a shared Author. The book *The Fellowship of the Ring* has John Tolkien as its writer, is categorized with the *high fantasy*-genre and is sold for a price of €22,99. Since the bidirectional reference exists, the second book can also be linked to the same Author, however the *The Silmarillion* does have different values for its *genre* and *price*, respectively *mythopoeia* and €10,49.

The concept of meta-modelling can be viewed as a multilayer process, every layer below being a instance of the current model. These layers are numbered and flow from the M0 layer which is the real world example, M1 is the primary model and so on.

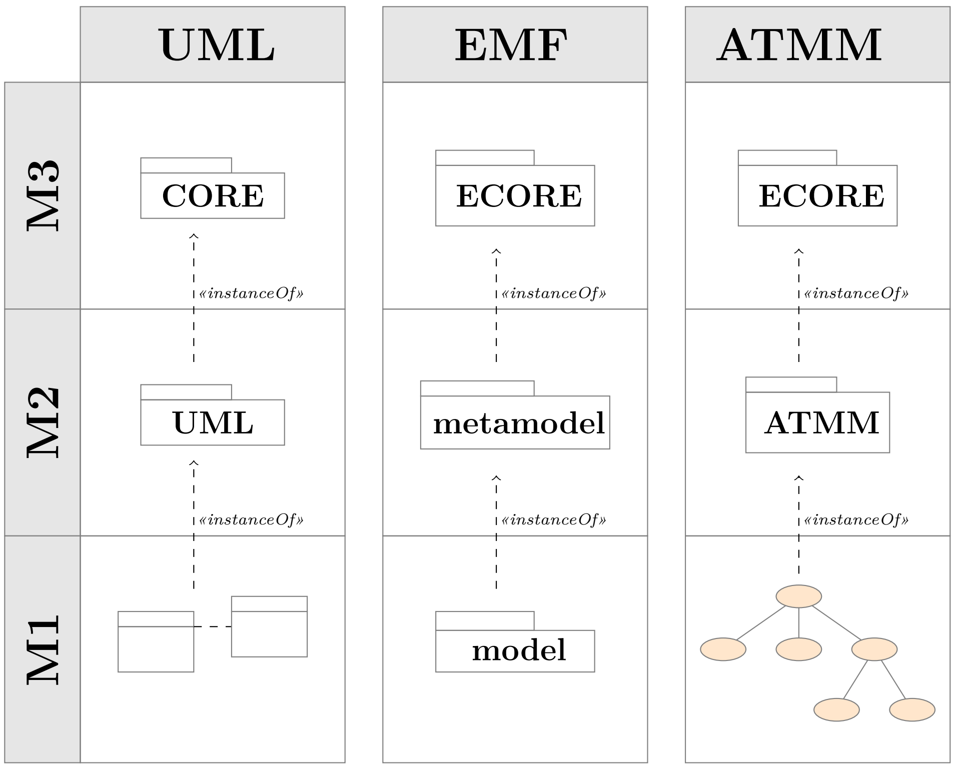


Figure 4.2: M1,M2,M3 metamoddeling example

An example of the layered concept meta-modeling works can be found in Figure 4.2. The left column displays how a class diagram (M1) modelling actual code is an instance of UML, which in turn is modelled in the model for UML. This second tier model (M2) is considered the meta-model of the original code.

For our AT scenario (right column in Figure 4.2) we use a similar scenario, we first model a real-life situation into an AT (M1), and make sure that this AT conforms to our M2 model (meta-model). And we have made sure that the meta-model is compliant with Ecore (M3).

Major advantages of using this concept were identified and are quickly discussed below.

Transference

The ability to transfer a model from one system into another is a major requirement when working in big projects and multiple teams. Clear constraints must be enforces to ensure that models are transferable and using a meta-model does exactly that. It enforced the user to conform to an agreed upon standard.

Correctness

When sharing models one of the most important parts is that the model is correct. One way to ensure this is to put restrictions on the model and make sure developers define the instantiations correctly. The whole meta-modeling concept (if used correctly) will make identification of mistakes which are not conforming to the meta-model easy.

Clarification

When discussing models with people from different departments using an agreed upon standard will help avoid misunderstandings. The meta-model will make it possible to synchronize similar items from different domains based on predefined rules, e.g. a *customer* in the sales department might be combined with *clients* from the legal department.

Compactness

Using the concept of a meta-model as an intermediate model will greatly limit the amount of transformations needed if the amount of input / output languages increases. As can be seen in the table, given a number of input languages (n_{in}) and a number of output languages (n_{out}) the amount of languages follows a linear scale ($n_{in} + n_{out}$) instead of exponential ($n_{in} * n_{out}$).

Amount of transformations needed			
n_{in}	n_{out}	$\#_{without\,MM}$	$\#_{with\,MM}$
1	1	1	2
2	2	4	4
3	3	9	6
20	20	400	40

Restrictions

One limitation that is frequently mentioned by users is that a meta-model enforces limits the design options a developer has. It is of vital importance that during the creation of the meta-model a balance is found between its completeness and its restrictiveness.

Importance of the design choices

Making the correct design choices is by far the most important step in creating a meta-model, hence this step often takes up more than 60% of the required time. A lot of meetings with domain experts is needed before a consensus has been achieved and even then, the best meta-models will have their limitations.

The design process for our Attack Tree Meta Model (ATMM) has undergone a number of iterations every time refining the model and adding or removing special features, ending up with a meta-model, whose 1.00 version will be discussed in the next section.

4.2 Attack Tree Meta-Model

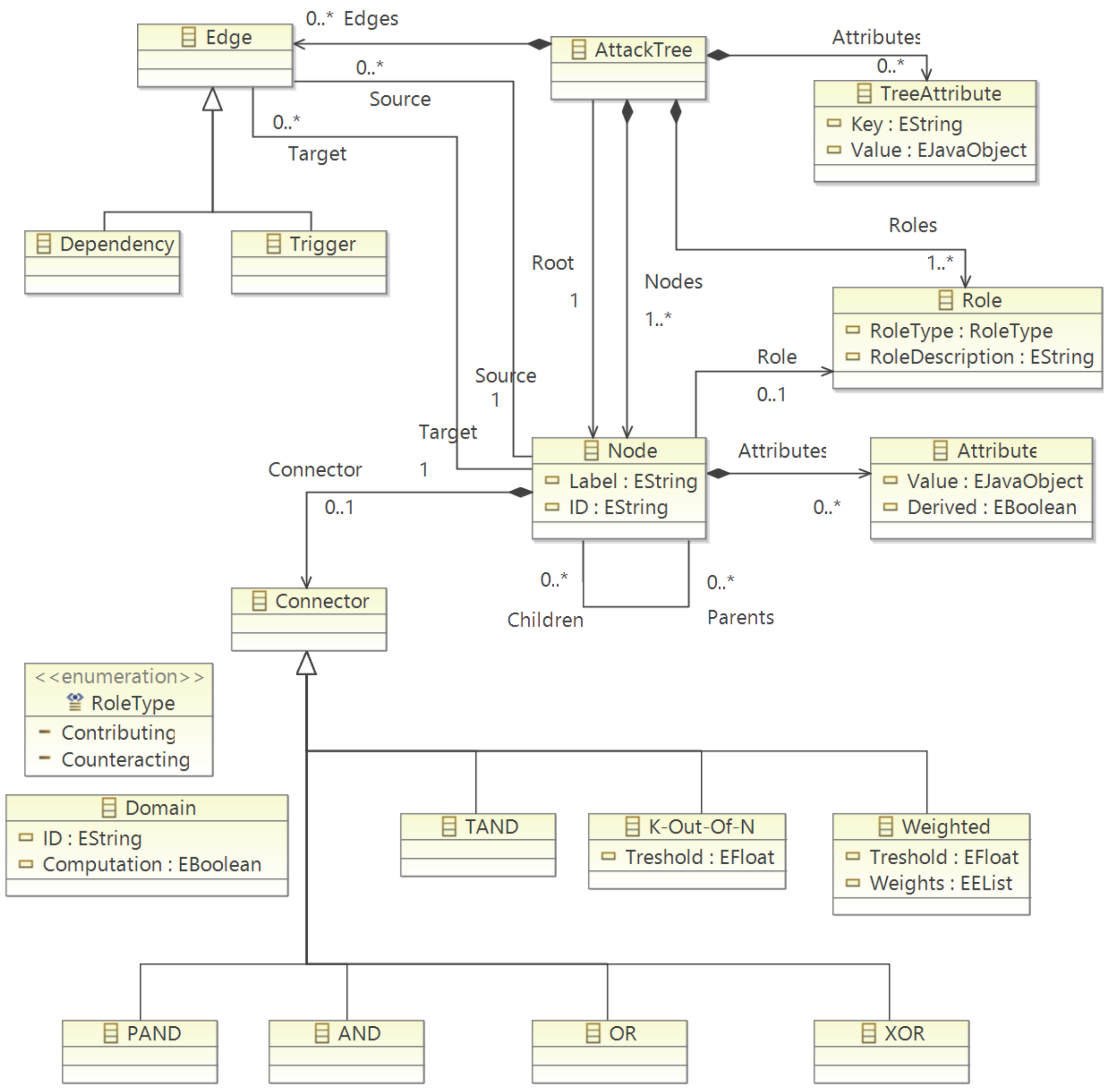


Figure 4.3: The proposed meta-model v1.0

This section will address the proposed meta-model, the final result can be found in Figure **??** and in Appendix F. This meta-model has been created in collaboration with D. Huistra, all the features used for the Uppaal transformation originate from this collaboration. Huistra included further refinements to include for this work less relevant features of ATs . The upcoming subsections will address the major parts of the meta-model individually. At first, it will address the basis / core of the meta-model and how it can represent the basics of every attack tree. Secondly, a subsection will be dedicated to the choices made regarding the different types of connectors. Attributes will be addressed in the third subsection and finally a subsection will quickly discuss roles and edges.

4.2.1 Basis / Core

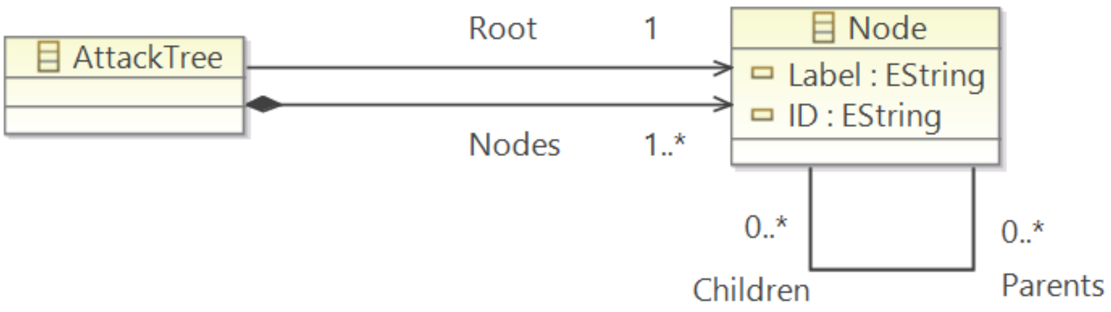


Figure 4.4: Core of the meta-model

As can be seen in Figure 4.4 the main parts of the meta-model consist of the tree representation. One *Attack Tree* has one *root*, which is an instance of a *Node*. In turn, each *Node* has zero or more *children*, these *children* have an inverse link to their *parent(s)*. A *Node* can be one of two possible nodes, either it has no *children*, which makes it a *leaf* and allows it to have attributes, or it does have one or more *children* which makes it an intermediate node and requires a connector to be specified [1]. More about the different kind of connectors in the upcoming section, but for the representations of basic Attack Trees only the AND and OR are required.

[1]note that these restrictions are not enforced by the meta-model itself, but can be enforced by the EVL validation set [26][27]

4.2.2 Connectors

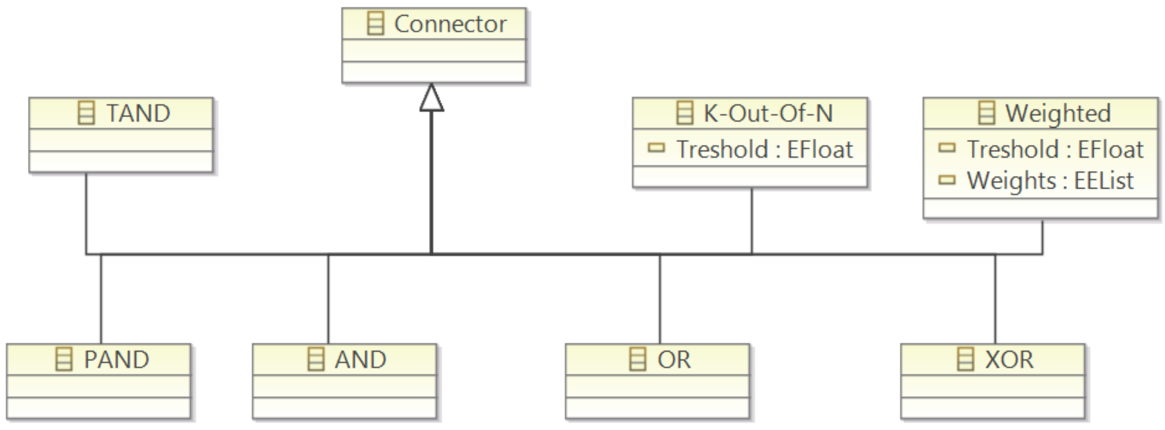


Figure 4.5: Connectors used for the meta-model

The meta-model provides a set of predefined *Connectors*, besides the basic *AND* and *OR* the following extra *Connectors* are defined.

- **XOR**. Exclusive OR, only one of its children can be triggered (where an ordinary OR accepts multiple children to be triggered).

- **PAND**. Enforces order in the execution of its children: a child cannot finish before its previous sibling has been completed. Also called SEQ or Ordered-AND this connector enforces order in the the execution of its children. [2]

- **TAND**. A normal AND, but enforced a timed delay between the children.

- **K-out-of-N**. This feature can be found in multiple extensions of Attack Trees, it represents a gate which activates if K out of the total N children have been triggered. Note that this Connector has an variable to store its *Threshold*

- **Weighted**. OWA trees [47] use this to distinguish between the importance of different children.

[2]the main difference between this and PAND is that SAND can only start if its previous sibling has been completed, while the PAND only enforces order in the finishing.

4.2.3 Attributes

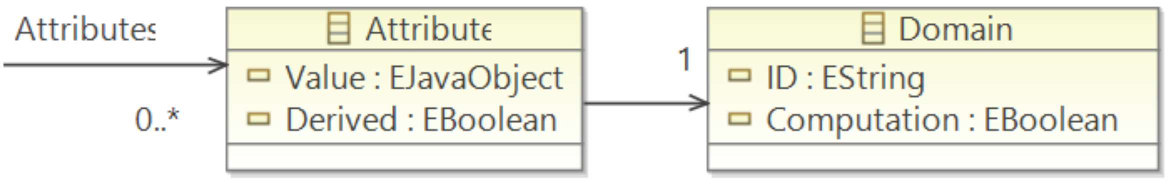


Figure 4.6: Attributes in the meta-model

Node Attributes are an important feature in the analysis of Attack Trees and proved difficult to define very broadly. For every limitation the meta-model enforced it was easy to find alternatives that could not be modelled. Therefore the ATMM ended up with an easily adaptable solution. Every *Attribute* has one domain, representing the way this variable should be treated in analysis methods. How these domains are defined is left to the users of the meta-model, e.g. a variable for the calculation of the total cost could belong to the addition-domain, enforcing an addition of the costs if the nodes are triggered. While a probability variable might belong to a multiplicative-domain, enforcing multiplication in outgoing transformations.

Furthermore every *Attribute* has its *value*, which is where it stores the actual number (or similar object) that is used for the calculation. To keep the meta-model generic this is defined as a *EJavaObject*, allowing for the value to be an instance of almost every type, e.g. the previously mentioned cost variable would be of type *Integer* and the probability would be of type *Double*.

4.2.4 Roles and Edges

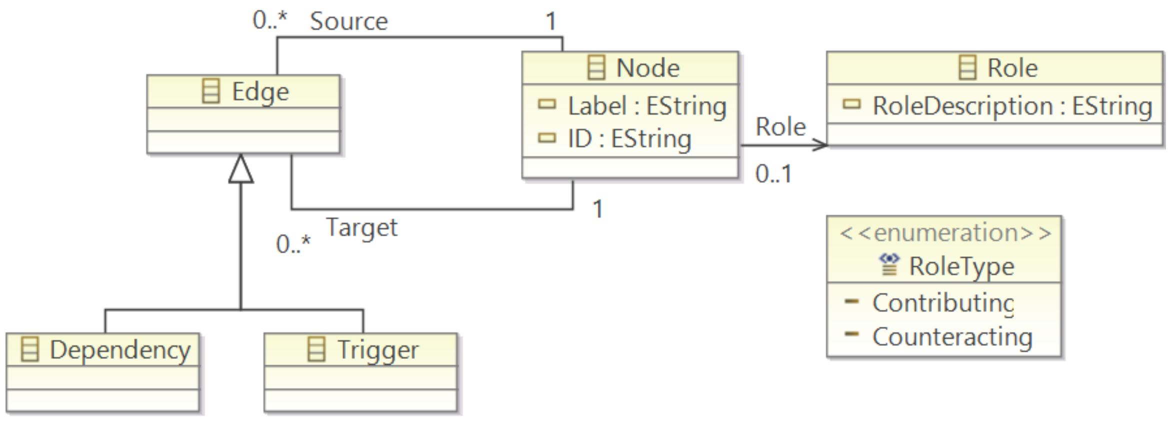


Figure 4.7: Roles and Edges in the meta-model

Roles

The meta-model defines two types of *Roles*, these define the high-level behaviour of nodes and can either be *contributing* or *counteracting*. This is where the definitions in the literature become a bit confusing depending on which implementations of Attack Trees is used. For this meta-model, the roles are defined as follows:

Definition 4.2.1. (Contributing) . A *Node* is considered *contributing* if it contributes to achieving the attackers goal. This can also be named *attacking*-node.

Definition 4.2.2. (Counteracting) . A *Node* is considered *counteracting* if it counteracts reaching the attackers goal. This can also be named a *defensive*-node.

There are quite a number of more specialized roles, i.e. [37] looks at this a bit different and includes failures. The meta-model allows for an additional *description*, to further separate different roles. As a design choice these were not defined as additional *RoleTypes* as the use of these extra roles is fairly limited.

Edges

For some functionality additional edges are needed on top of the normal edges between *Nodes*. This is required for the definition of *Trigger* and *Dependency* Edges.

Trigger Edge, a directional edge between two nodes, A and B. If A triggers, the trigger edge will also trigger B. Often these nodes are in different parts of the trees or can be used to trigger whole sub-trees at once.

Dependency Edge, a dependency edge between two nodes requires the incoming node to be dependent on the outgoing node. If nodes A and B are connected by a dependency edge, B can only be triggered if A has been triggered.

Example of an instance

In Figure 4.8 one can find an example for a simple attack tree and its ATMM instance. As can be seen, even small trees will become rather large instances. For future users however this is irrelevant as this model is an intermediate model and will not be visible in the whole transformation process

4.3 Design Choices & Limitations

Limitations

As the overview paper written by Kordy et. al in 2014 [32] has been used as a basis for the meta-model, all formalism that are represented in that paper have been considered when creating the ATMM. For all the missing or newer formalisms we have not evaluated their fit into our model. An overview of the formalisms that cannot be fully represented can be found in Table 4.1

Easy to implement extensions

Additional gates and connectors are fairly easy to be included in future versions, as soon as the need for such arises. For the simple gates the extension would only entail adding one additional instance below the *Connector*-class.

Extra roles and domains are also fairly straightforward to be included in extensions, as they only require adding an extra RoleType or Domain.

Hard to implement extensions

Every change that involves changing the basic structure of ATMM should be avoided wherever possible, e.g. allowing an additional root will destroy the basis concept of the attack tree, as the attack now has multiple goals which can be completed separately.

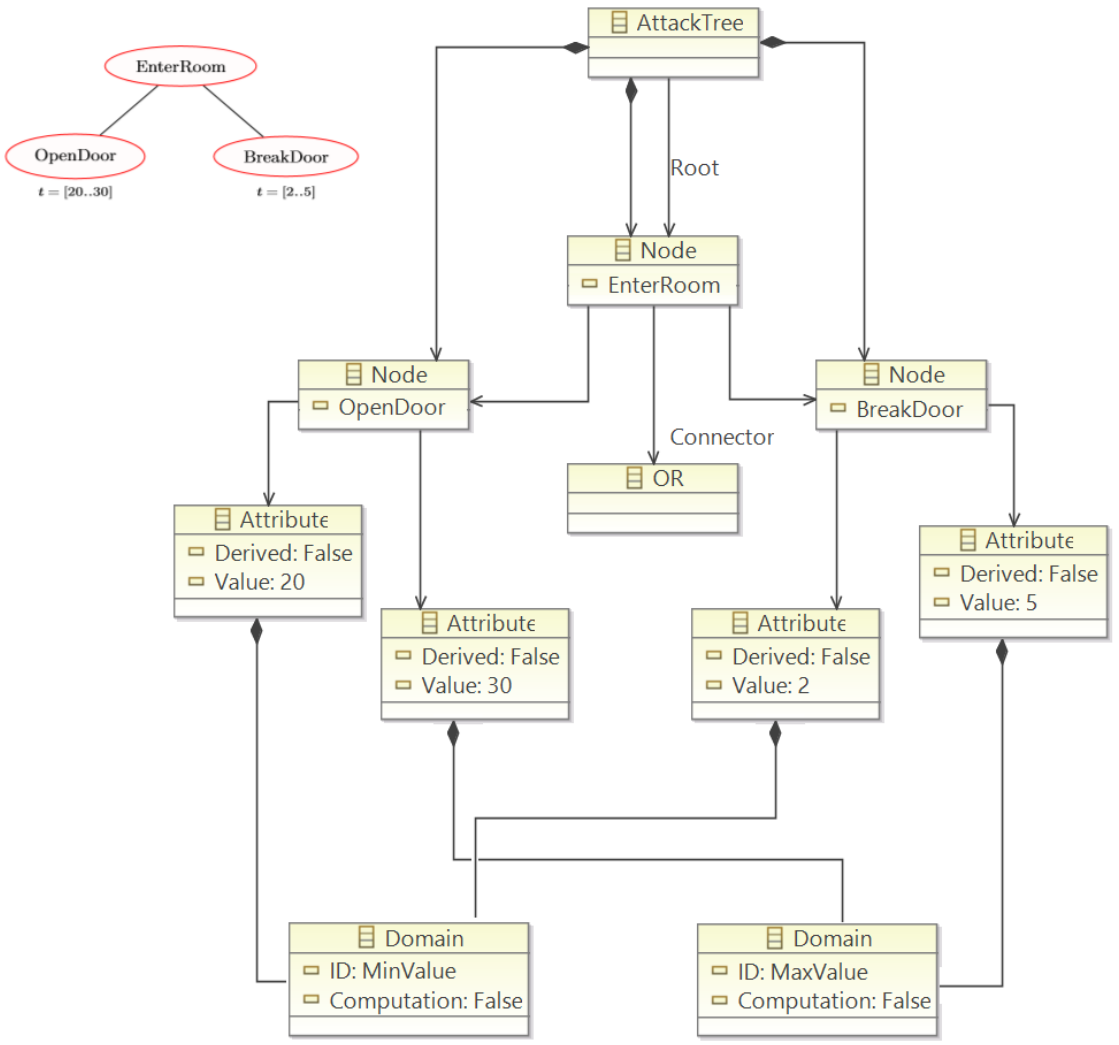


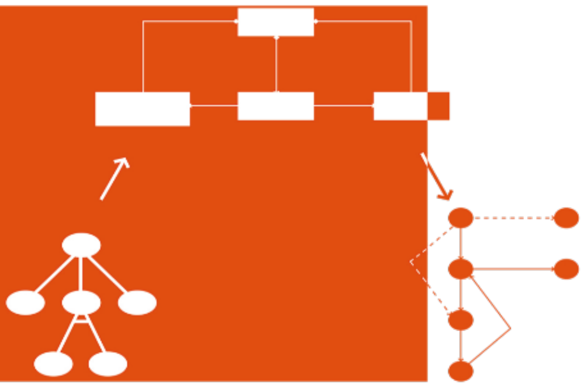
Figure 4.8: Example instance of a small Attack Tree

Formalism	Can be represented?	Comments
Fully Represented		
Attack Trees	Full	
Augmented attack trees	Full	
OWA trees	Full	
Vulnerability cause graphs	Full	
Parallel model for multi-parameter attack trees	Full	
Serial model for multi-paramter attack trees	Full	
Extened fault trees	Full	
Improved attack trees	Full	
Time-depdendent attack trees	Full	
Attack-defense trees	Full	
Attack-response trees	Full	
Defense trees	Full	
Unified parameterizable attack trees	Full	
Augmented vulnerability trees	Full	
Attack countermeasure trees	Full	
Protection trees	Full	
Partially Represented		
Fault trees for security	Partially	*Cannot represent multiple connectors*
Dynamic fault trees for security	Partially	*Cannot represent multiple connectors*
Security goal models	Partially	*Not all edges are available*
Insecurity flows	Similar	*Different structure*
Intrusion DAGs	Similar	*Multiple roots cannot be represented*
Security goal indicator trees	Partially	*Not all edges are available*

Table 4.1: Formalisms from Kordy et al. [32] and how they fit in the ATMM

Chapter 5

Transformations



5.1 Problem Statement

In previous chapters the basis has been constructed, Chapter 3 provided the end goal and Chapter 4 its intermediate meta-model. This Chapter will focus on the translations from one instance into another. Which direction to take with these translations depends on our intentions. The following selection criteria provided the basis for this choice:

Completeness . This can be assessed along multiple dimensions: it should be complete with respect to the needs of the user, complete in its analytical context and the transformation itself should be complete, e.g. the whole model is transformed, and the transformation is consistent.

Simplicity/Usability . For reasons of readability and extensibility having a simple and easy to understand transformation is important. A simple to use modeling method should be easy to install, easy to comprehend, efficient, easy to use and should guide the users when errors occur. Since the goal of this work focuses on laying the foundations for future re-usability and extensive improvement, we consider simplicity as one of the top priorities.

M2T Support . Since our final model has its instances in a xml-format it is very important that the transformation has the option for model-to-text (M2T) transformations

Scalability . The ability to handle systems of arbitrary size without slowing down too much is one of the most important criteria if we look to the future work of this thesis: after its initial deploy it is planned to be used on large systems. A feature often mentioned in this context is a separation of concerns, the ability to work on isolated parts of a solutions instead of having to redesign it as a whole.

Documentation . The availability of proper documentation and examples will obviously be considered an desirable and advantageous feature.

Available knowledge . Lastly the amount of available knowledge is taken into account, either by different sources within the FMT-chair or from courses given on the University.

The goals of this chapter are as follows:

◇ Define a transformation from ADTool-xml into the meta-model.

◇ Define a transformation from the ATMM into the ADTool-xml.

◇ Define a transformation from the ATMM into the Uppaal-model defined in chapter one.

5.2 Epsilon Language Family

Epsilon is a Model Management Platform developed by the University of York. This platform provides multiple task specific languages, among which are the Epsilon Transformation Language (ETL), Object Constraint Language(OCL), Epsilon Validation Language (EVL), Epsilon Generation Language (EGL) and Epsilon Merging Language(EML). All these different languages are implemented in components on top of the Epsilon Object Language (EOL), which forms the basis for the whole Epsilon Package integrated in Eclipse [28].

For our application the first three components mentioned are the most relevant i.e. ETL, EVL and EGL.

ETL is used for model to model transformations, it is a hybrid model transformation language and is integrated in Epsilon, which provides seamless communication with other task specific options. [38]

The transformation can either originate from xml or ecore models and also generate an xml or ecore model. In our approach it is used in three separate transformations.

- From ADTool into UATMM
- From UATMM into ADTool
- From ADTool into the Uppaal.ecore model

EVL is used for validation purposes and can ensure that models remain valid and consistent after multiple transformation, e.g. it can be used to test if an Attack Tree always has a root. Within our implementation this is used for the outoging transfromations of the ATMM.

Finally EGL is a template based model-to-text transformation language that can be used to generate text from models. Placeholders can be placed in between large predefined pieces of text, this makes swapping templates (either EGL-templates or Uppaal-templates) in and out fairly easy.

ETL example

```
<adtree>
  ...
  <domain id="MinCost1">
    <class>lu.uni.adtool
.domains.predefined.MinCost
    </class>
    <tool>ADTool</tool>
  </domain>
  <domain id="ProbSucc2">
    <class>lu.uni.adtool
.domains.predefined.ProbSucc
    </class>
    <tool>ADTool</tool>
  </domain>
</adtree>
```

```
// Transform all domains
@primary
rule ADToolDomain2UATDomain
        transform ad :
           ADTool!t_domain
        to gad :
           UATMM!Domain {

        gad.ID = ad.a_id;
        gad.Computation =
           true;
}
```

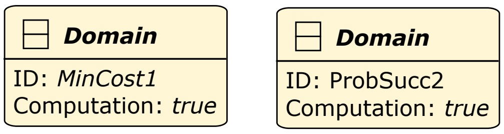


Figure 5.1: Resulting instance

The above example transforms the xml domains in the ADTool format (*left*) into the instance of ATMM (*bottom right*) using a snippet of the actual transformation (*top right*). As this is for demonstration purposes only we used the easiest part of the transformation. The *ADToolDomain2UATDomain* rule has the following general behavior, for every tag "domain" in the input model ADTool (*ADTool!t_domain*) it will create Domain instances in the ATMM output model (*UATMM!Domain*). And for every newly created Domain it will set the values as ID and Computation. Since Computation is not relevant for the ADTool transformation, but is present in the meta-model for different formalisms, it will be always set to *true*. The ID however is relevant and will be set to *MinCost1* and *ProbSucc2* accordingly. This already concludes this transformation. For the rest of the information, the *<class>* and *<tool>* tags does not require storage in the meta-model can be generated when a reverse transformation is executed. The class element is simply a package with the domain and the tool is always adtool as there is a unique transformation.

EGL example

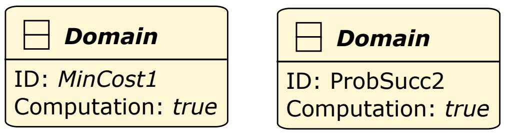


Figure 5.2: Input instance

```
[%
// EGL transformation
%]
<html>
<head>
  <title>[%=Domain.allInstances().size()
     %]
  domains</title>
</head>
<body>
<h1>[%=Domain.allInstances().size()
   %]
domains</h1>
<table>
[% for (domain in
   Domain.allInstances()){%]
 <tr><td>[%=domain.ID
    %]</td>
<td>[%=domain.Computation
   %] </td></tr>
[% } %]
</table>
</body>
</html>
```

```
<html>
<head>
  <title>2 domains</title>
</head>
<body>
  <h1>2 domains</h1>
  <table>
    <tr><td>MinCost1</td>
    <td>true</td></tr>
    <tr><td>ProbSucc2</td>
    <td>true</td></tr>
  </table>
</body>
</html>
```

This *EGL* example starts from the instance we created with the *ETL* example. It has two Domains which both have an ID and a Computation value. Our goal is to transform this into something text based. Since the domains do not directly transform to something textual in our Uppaal model, we created this documentation generator purely for demonstration purposes. Based on the number of available Domains, this translation will generate a web-page. The input model can be found in the top left, the transformation on the right and the result in the bottom left of the example. The final result creates a HTML page, gives it a title depending on the number of Domains (*Domain.allInstances().size()*), it also sets this same value as an H1 header on the page. Furthermore the page contains a table element, this table is generated based on the information within the domains. For each instance in Domain.allIstances() a row is created with the ID and the value of its Computation variable respectively. This approach is similarly used within the Uppaal transformation as it will generate XML code, which can be loaded into Uppaal.

5.3 Transformation choices

The first transformation that will be presented is the transformation from an example ADTool *attack tree* stored in its xml format (Figure 6.3). This transformation will eventually lead to the ATMM instance in a *<name>.model* file, the result of our transformed example can be found in Appendix REF.

As the transformation is based on ETL, it contains **pre** and **post** sections, which are two special sections which are executed respectively before and after the actual transformation rules.

```
pre {

        var javaDateBegin = new Native("java.util.Date");
        javaDateBegin.toString().println("> Starting
           ADTool2UATMM transformation: ");

        // Start a recursive XML Teardown to determine
           parent/children relations.
        // (Operation instead of rules, as rules did not seem
           suitable)
        //ADTool.root.e_node.recursiveXMLTeardown(true, null);
}

post {
        var javaDateEnd = new Native("java.util.Date");
        javaDateEnd.toString().println("> Completed
           ADTool2UATMM transformation: ");
        var elapsed = javaDateEnd.getTime() -
           javaDateBegin.getTime();
        var minutes = elapsed / 60000;
        var rest = elapsed.mod(60000);
        var seconds = rest / 1000;
        var mili = rest.mod(1000);
        System.out.print("> Time elapsed: ");
        if (minutes <> 0) System.out.print( minutes + " min,
           ");
        if (seconds <> 0) System.out.print( seconds + " sec :
           ");
        System.out.println( mili+ " ms" );
}
```

pre / post

The **pre**-section only stores the current time stamps and prints it to the console. This gives a good indication regarding the total run time of the program and is used for measurements. The **post**-section does something similar, but also calculates the difference and prints this in user friendly output format.

Other than transformation rules ETL allows for defining our own functions, called operations, which can greatly benefit the transformation process.

Operations

```
@cached
operation Any getAttackTree() : UATMM!AttackTree{
        return UATMM!AttackTree.allInstances.last();
}


operation Integer mod(i : Integer) {
                return self - (self/i * i);
        }
```

The above operations illustrate what operations within ETL are primarily used for. First they are used for removing duplicate and redundant code from the main transformation, as can be seen in *getAttackTree()*, which retrieves the overall instance of *AttackTree* (of which there is only one) . A disadvantage of the way this operation is implemented is that it uses the very expensive .allInstances functionality. Especially in larger model instances, this function will severely slow down the application. To avoid this slow down, this operation is only executed once, enforced by the *@cached* annotation, and afterwards directly return its cached result. This simple operation provides a speed-up in the application, already observable in smaller models.

The second operation is an operation that extends the options offered by ETL, in this case there was no mod functionality available, so we created our own mod functionality.

Primary rules

```
// Transform all domains and set several attributes by
   obtaining
// info from attributes and child nodes
@primary
rule ADToolDomain2UATDomain
        transform ad : ADTool!t_domain
        to gad : UATMM!Domain {

        gad.ID = ad.a_id;
        gad.Computation = true;
}
```

Since ETL is a rule based language, the majority of its transformations will be done by rules. Two of these type of rules are the primary rules and lazy rules. All primary rules will be executed before any of the other rules, whereas the lazy rules are only being executed if they are needed. To activate lazy rules *Epsilon* provides the equivalent() function.

In this transformation primary rules are used because of the importance of the domains. All the domains need to be created before the logic of the *recursiveXMLTeardown()* can correctly link them. The function recursiveXMLTeardown() itself is called upon the root node of the XML document and recursively called upon its children, to handle the nodes in a correct order to create the correct parent / child links. This recursive part is used to maintain the tree structure within in the ATMM. Even though the whole concept of Epsilon is very interesting and most of the transformations can (arguably even better) be run in the developers environment of Eclipse, for completeness the transformation and its standalone version have been included in a Java program that can run outside of eclipse. This was not as straight-forward as initially expected, even though there was standalone support on the Epsilon website [26], the Java code didn't seem to work inside a Java Archive (jar-file). The reference to the meta-data (i.e.

ETL-files) ran into the *"URI is not hierarchical"* exception, this seemed to be an error which originated from within some epsilon source files which we could not change.

However an update changed the way these files are read and now the jar, after some URI / URL conversions is fully functional and ready for use.

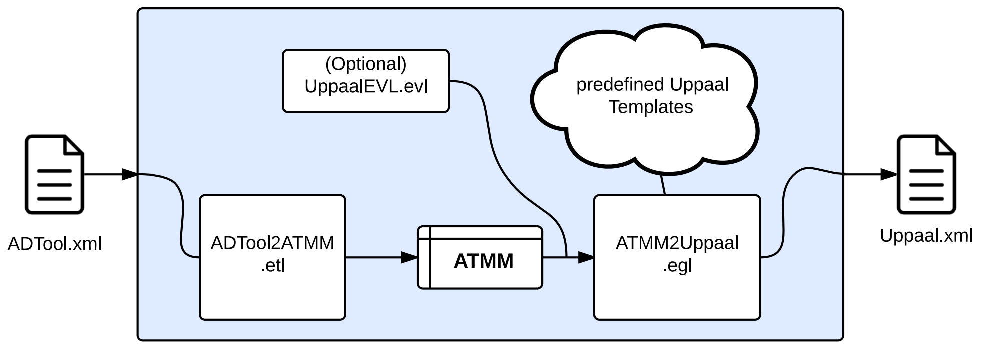


Figure 5.3: Basic chain within the java program

This led us to the following simplified extension of the *EpsilonStandaloneExample*, a general program run performs the following tasks:

- the program takes the URL of the file as an argument.
- retrieves the URI from the URL.
- Phase 1
  - loads the ETL model.
  - loads the input file.
  - performs transformation.
  - stores in intermediate model.
- Phase 2
  - loads the EGL models.
  - loads the intermediate model.
  - performs transformation.

– outputs the final result into the output file.

The following code is an example of a standalone java application. Note that this code is not my final code, but simply an example of how a java application can load and execute the templates.

```
package egl;
import *;

public class EglStandalone extends EpsilonStandaloneExample {

        private String output_url = "*/output/uppaal.xml";
        private boolean etl = false;
        private String input_url;

        public static void main(String[] args) throws Exception {
                if(args.length > 0) {
                        new EglStandalone(args[0]);
                } else {
                        System.out.println("Usage UATMM2Uppaal <input_file>");
                }
        }

        public EglStandalone(String input) {
                input_url = input;
                try {
                        System.out.println("");
                        setSource("transformation/ADTool2UATMM.etl");
                        this.execute();
                        etl = false;
                        System.out.println("Intermediate Model stored in:
                            model/Instance.model");
                        setSource("predefined/template.egl");
                        this.execute();
                } catch (Exception e) {
                        System.out.println("Some error occurred during the reading
                            of the files");
                        e.printStackTrace();
                }
        }

        @Override
        public IEolExecutableModule createModule() {
                if (etl) {
                        return new EtlModule();
                } else {
                        return new EglTemplateFactoryModuleAdapter(new
                            EglTemplateFactory());
                }
        }

        @Override
        public List<IModel> getModels() throws Exception {
                List<IModel> models = new ArrayList<IModel>();
                if (etl) {
                        models.add(createXmlModel("ADTool", input_url));
                        models.add(createEmfModel("UATMM", "model/Instance.model",
                            "model/UATMM.ecore", false, true));
                } else {
                        models.add(createEmfModel("AT",
                            "model/Complete_Tree_MM_Instance.model",
                            "model/UATMM.ecore", true, false)); //DK
                }
                return models;
        }

        @Override
        public String getSource()  {
                return resource;
        }

        @Override
        public void setSource(String s)  {
                resource = s;
        }

        @Override
        public void postProcess() {
                if (!etl) {
                        PrintWriter writer;
                        try {
                                writer = new PrintWriter(output_url, "UTF-8");
                                writer.println(result);
                                writer.close();
                        } catch (FileNotFoundException |
                            UnsupportedEncodingException e) {
                                System.out.println("The file required cannot be
                                    found or has an incorrect encoding");
                                e.printStackTrace();
                        }
                }
        }
}
```

5.3.1 Java program Adt2Upp.jar

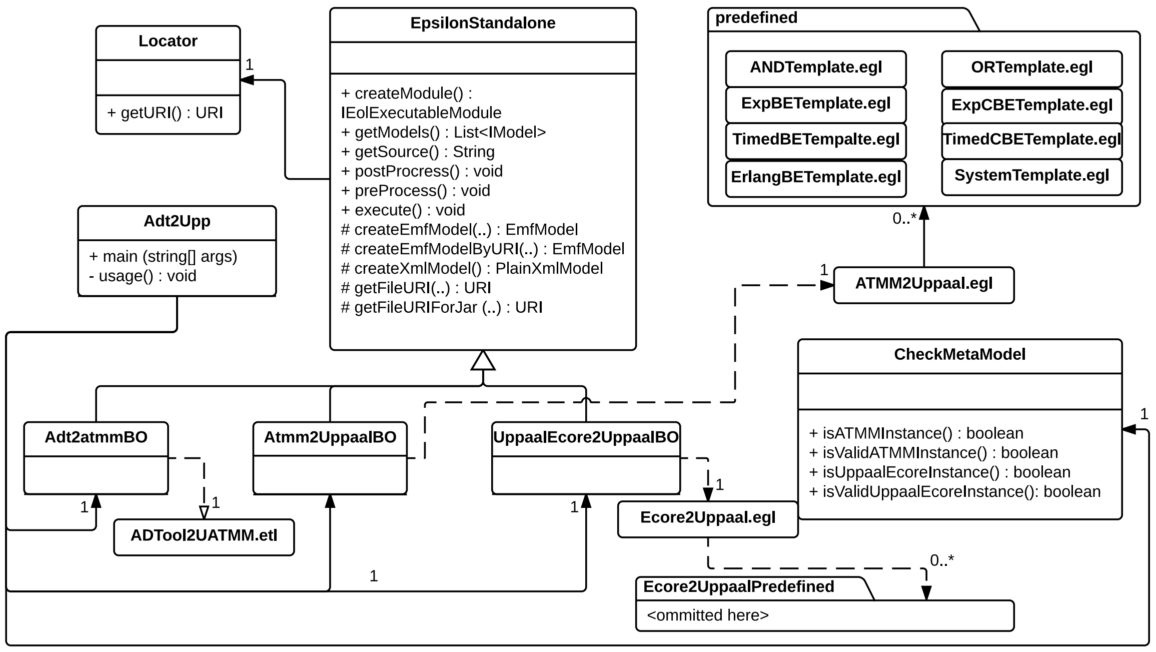


Figure 5.4: Class diagram of Adt2Upp.jar

The final code can be found on the github of the FMT-faculty[1].

In Figure 5.4 we have the class diagram of the java program delivered with this thesis, the central hub in this class diagram is the java class *Adt2Upp*, this class contains the main function and has the links to three *Business Objects (BO)*, this term is a part of the software patern (Core J2EE Pattern). These BOs are responsible for providing the logic of the transformation. The classes *Adt2atmmBO*, *ATMM2UppaalBO* and *UppaalEcore2UppaalBO* are all abstractions from EpsilonStandalone and only differ in there settings. Every class has couple of functions overwritten from its abstraction, it depends slightly which functions, but for example every class needs to have *getSource()* and *getModels()* overwritten.

Furthermore in the class diagram its is clear which transformation templates are load as a default, for Adt2attmBO this is *ADTool.etl*, for Atmm2UppaalBO this is *ATMM2Uppaal.egl* and finally for UppaalEcore2UppaalBO this is *Ecore2Uppaal.egl*.

[1]https://github.com/utwente-fmt/Adt2Upp

Both the EGL transformations have many imported EGL files, for *ATMM2Uppaal.egl* they are included in the predined package. For readability the imports for *Ecore2Uppaal.egl* have been omitted as there are many. For an example EGL template we refer the reader to Appendix D, which will contain the code *SystemTemplate.egl*, as this is a completely static template, this is a great illustration how close the EGL code refers to the Uppaal input XML.

5.3.2 ADTool2ATMM

A ETL transformation that requires the XML export from ADTool to be converted into an instance of the meta-model presented in chapter 3. As a stand-alone transformation this etl-file can be run from inside the development environment using the following settings.

The code from adtool2atmm.etl can be found in Appendix A

5.3.3 ATMM2ADTool

A ETL transformation that can be used to reverse transform the instance of ATMM into a XML file in the correct structure to be imported back into ADTool.

The code from atmm2adtool.etl can be found in Appendix B

5.3.4 ATMM2Uppaal

Finally the third transformation is the EGL transformation that transforms an instance of ATMM into a working Uppaal model on which the required analysis can be executed.

The EGL code can be found in Appendix C

5.3.5 Compiled Jar

Compiling the whole java program and exporting it as a jar package will give the user a standalone tool (Adt2Upp.jar) which can be used as a command line tool to transform ADTool XML into Uppaal output.

```
Usage:> java -jar Adt2Upp.jar [options] <input_file> <output_file>
input_file:
   *.xml    : as ADTool input
    *.model  : as ATMM input
Options:
   -skipEVL : Skipping the Epsilon Validation Step
    -noEGL   : Skipping the Epsilon Generation Step, producing model instance
```

Chapter 6
Transformation Demonstration

This Chapter will demonstrate the use of the *Attack Trees* from Chapter 2 and run through the process of one transformation run. During the run they will be transformed into an instance of the *Attack Tree Meta-Model* from Chapter 4. Both from and towards the ATMM the transformations provided in Chapter 5 will be used. Finally the result be the Uppaal model desired by the process, it will contain some of the templates provided in Chapter 3, thereby linking all the chapters as one big chain and perfectly demonstrating how the transformation tool works.

On the next two pages the Attack Tree will be introduced and its XML code will be shown on the right page. Similarly the two pages after that will present the Attack Tree as an instance of the meta-model on the right page and use the left page to properly discuss how the transformation used the XML code from ADTool to provide this instance. Finally four pages will dedicated to show the five templates used and show how they are linked together through the declaration part of the Uppaal model and the System part.

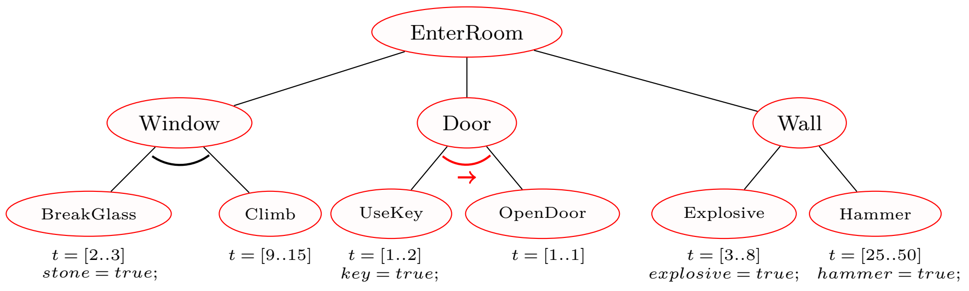

EnterRoom
Window
Door
Wall
→
BreakGlass
Climb
UseKey
OpenDoor
Explosive
Hammer
$t = [2..3]$ $stone = true;$
$t = [9..15]$
$t = [1..2]$ $key = true;$
$t = [1..1]$
$t = [3..8]$ $explosive = true;$
$t = [25..50]$ $hammer = true;$

Figure 6.1: EnterRoom Example

Modeling Example 1 in ADTool

Figure 6.1 displays the example that was introduced and explained in Chapter 2. In this section this AT has to be modeled in ADTool.

Firstly we create the structure, a disjunctive *root* with two conjunctive and one disjunctive children, each having two children. The last six nodes do not further refinements, making them *leaves*.

Secondly the *attributes* can be quickly separated into two categories, *time* and *booleans*. For the *time* there are two separate *domains*, the minimum times (*2, 9, 1, 1, 3, 25*) and the maximum times (*3, 15, 2, 1, 8, 50*). For this we used a *domain* provided by ADTool *MinTimeSeq* (Minimum Sequential Time).For the *booleans* we actually need 4 separate *domains*, because if an node for example does not have a key, the value is false (these are excluded from the image). ADTool also provides a *domain* for this, the SatScenario (*Satisfiability Scenarios*). All the 6 domains can be found on the bottom of XML code on the right page. Furthermore the XML maintains the tree structure and has 6 parameters for every leaf.

Things omitted for readability, every parameter for a *SatScenario* which has *false* as its value and the content of the *domain* nodes.

```xml
<?xml version="1.0" encoding="UTF-8"?><adtree>
  <node refinement="disjunctive">
    <label>EnterRoom</label>
    <node refinement="conjunctive">
      <label>Window</label>
      <node refinement="disjunctive">
        <label>BreakGlass</label>
        <parameter domainId="SatScenario1">true</parameter>
        <parameter domainId="MinTimeSeq5">2.0</parameter>
        <parameter domainId="MinTimeSeq6">3.0</parameter>
      </node>
      <node refinement="disjunctive">
        <label>Climb</label>
        <parameter domainId="MinTimeSeq5">9.0</parameter>
        <parameter domainId="MinTimeSeq6">15.0</parameter>
      </node>
    </node>
    <node refinement="conjunctive">
      <label>Door</label>
      <node refinement="disjunctive">
        <label>UseKey</label>
        <parameter domainId="SatScenario2">true</parameter>
        <parameter domainId="MinTimeSeq5">1.0</parameter>
        <parameter domainId="MinTimeSeq6">2.0</parameter>
      </node>
      <node refinement="disjunctive">
        <label>OpenDoor</label>
        <parameter domainId="MinTimeSeq5">1.0</parameter>
        <parameter domainId="MinTimeSeq6">2.0</parameter>
      </node>
    </node>
    <node refinement="disjunctive">
      <label>Wall</label>
      <node refinement="disjunctive">
        <label>Explosive</label>
        <parameter domainId="SatScenario4">true</parameter>
        <parameter domainId="MinTimeSeq5">3.0</parameter>
        <parameter domainId="MinTimeSeq6">8.0</parameter>
      </node>
      <node refinement="disjunctive">
        <label>Hammer</label>
        <parameter domainId="SatScenario3">true</parameter>
        <parameter domainId="MinTimeSeq5">25.0</parameter>
        <parameter domainId="MinTimeSeq6">50.0</parameter>
      </node>
    </node>
  </node>
  <domain id="SatScenario1">  ...
  <domain id="SatScenario2">  ...
  <domain id="SatScenario3">  ...
  <domain id="SatScenario4">  ...
  <domain id="MinTimeSeq5">  ...
  <domain id="MinTimeSeq6">  ...
</adtree>
```

ADTool-to-ATMM

Transforming the *AT* from Figure 6.1 into the *ATMM* will be done by the *ADTool2ATMM* transformation. This conversion is the first step in the complete chain and will result in an instance of the meta-model, Picture 6.2 contains the resulting instance.

The 6 *Domains* found in the XML representation (previous page) of the AT can be found at the bottom and are used to link to the *Attributes*. The labels of the AT directly corespondent to the *Nodes* in the Instance. To provide structure the nodes have been rearranged and the attributes have been grouped.

For simplicity and readability some features have been omitted. For example every *Node* has a link to the *AttackTree*, but all these edges have been removed, the only link that remains the *root*-link. Also every node has an ID but this is omitted, since only the *Label* is used in this process. Similarly the *derived* boolean in the *Attribute* is excluded in the Figure. Finally, every *Attribute* is an element of a *Domain*, but all these edges have been replaced with numbers, which in turn refer to the *Domain*. i.e. The attributes of the Node *BreakGlass* belong respectively to *MinTimeSeq5*, *MinTimeSeq6* and *SatScenario1*.

After the rearrangement of the nodes in the *ATMM* instance the tree structure of the AT in Figure 6.1 is still strongly recognizable. This is important because the eventual UPPAAL model should maintain the same structure and having this structure simplifies and guides the *model-to-text* transformation.

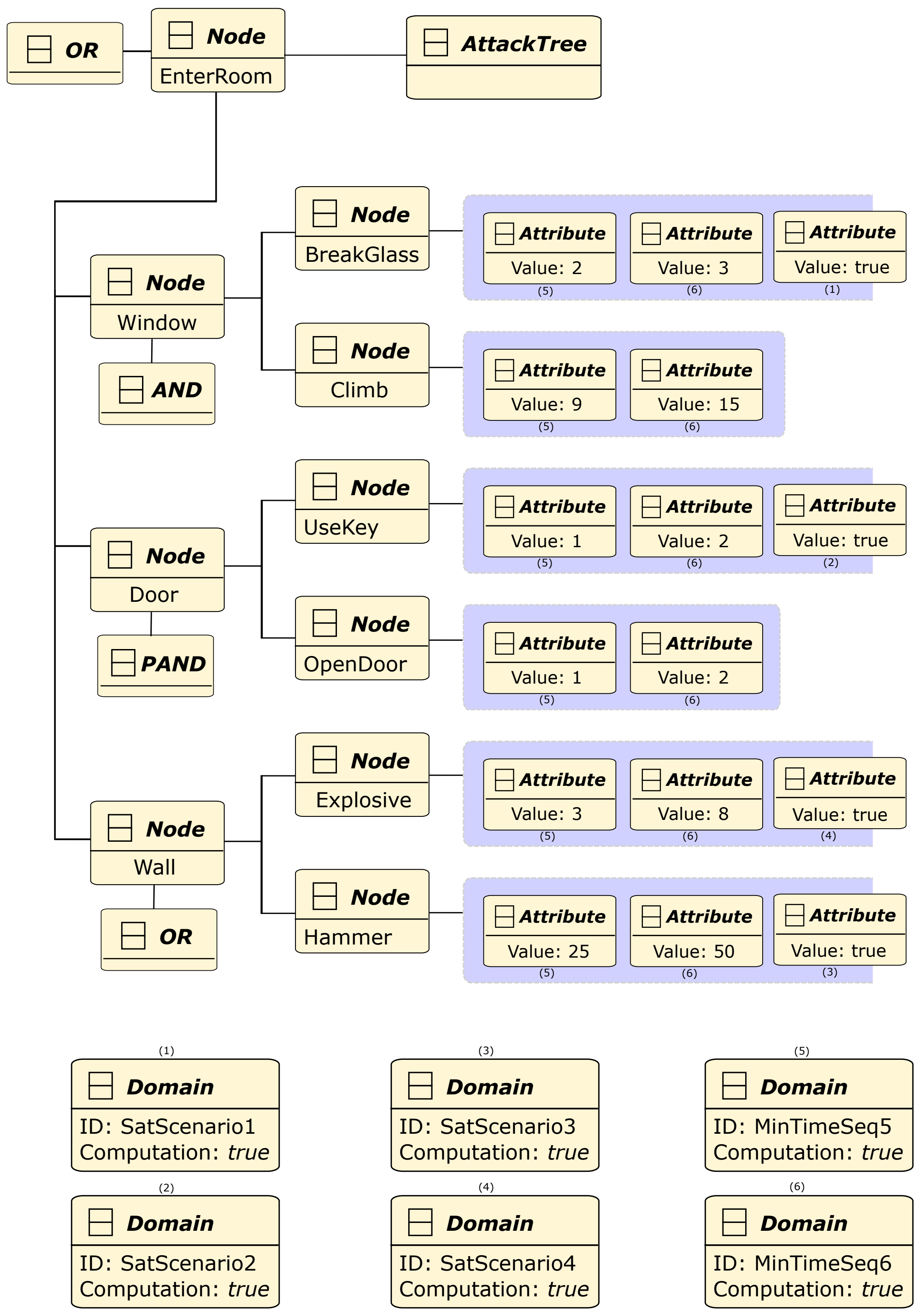


Figure 6.2: Example 1 as an instance of ATMM

The result of the *ATMM2Uppaal* transformation is displayed on the following three pages. The result needs five templates represent the AT presented in Figure 6.1. Respectively *TimedBE*, *TimedCBE*, *AND2*, *OR2* and *OR3*. The first two are the templates that are being used for the leaves, five leaves will have a normal *Timed Basic Event* linked to them and one (*OpenDoor*) will have a *Dormant / Cold Timed Basic Event* representing him. These templates can be found on the right hand page.

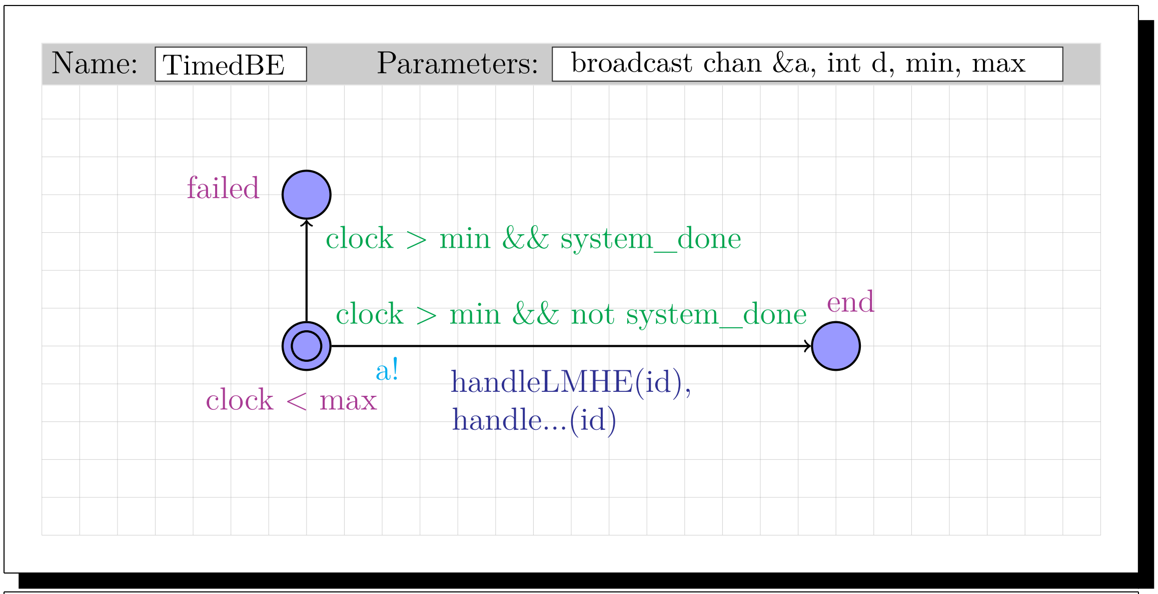
The other 3 templates are the templates for the intermediate leaves. Since we are representing SAND gates with a normal AND gates and are only linking the underlying leaves differently there is no need for a separate SAND gate. This leaves an *AND2* template that is used twice, and an *OR2* template and an *OR3* template that are used once.

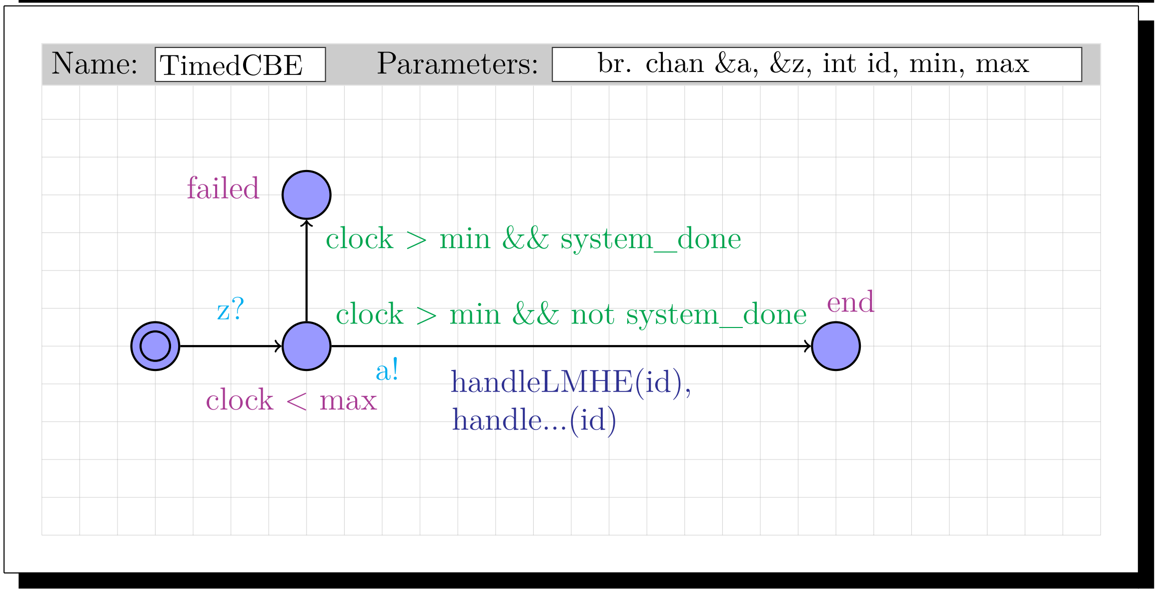
The generated code blocks can be found in Code 6.1 and Code 6.2, which respectively represents the code generated for the declaration part and the code that is generated for the system part.

The *declaration* part takes care of initialising all the required *broadcast channels*. Initialising the global variables that are needed, booleans *t*, *f*, *system_done* and *result_SatScenario3*. The last boolean is generated based on the available domains. Note that the timed domains do not need a global variable here because the timed templates take care of time. Further more the *SatScenario3* domains has an array list which stores all the values of the leaves, in this specific case the array has a length of size and contains *{true, false, true, false, true, true, }*. Last the generation defined the *handleSatScenario()* function, a function that is called from the activated leaves and contains the leaf-id to update the global *result_SatScenario3* based on the value at index id of the *SatScenario3* array.

The system part is used to create all the processes based on the templates defined earlier. As an example we pick the leaf *OpenDoor*, as mention above this leaf uses the *Dormant Timed Basic Element* template and is initialised with the following settings: *OpenDoor_success, UseKey_success, 1, 2, 3*. Respectively the channel it broadcasts on, the channel it waits for, the minimum time it waits, the maximum time it waits and the id for this specific leaf. Finally the system call parallels all the processes and this lets Uppaal know that all these processes need to be used.

Templates for the Leafs


Name: TimedBE
Parameters: broadcast chan &a, int d, min, max
failed
clock > min && system_done
clock > min && not system_done
end
a!
clock < max
handleLMHE(id),
handle...(id)


Name: TimedCBE
Parameters: br. chan &a, &z, int id, min, max
failed
clock > min && system_done
z?
clock > min && not system_done
end
a!
clock < max
handleLMHE(id),
handle...(id)

Generated code within the declaration tag

Code 6.1: Code generated for the declarations

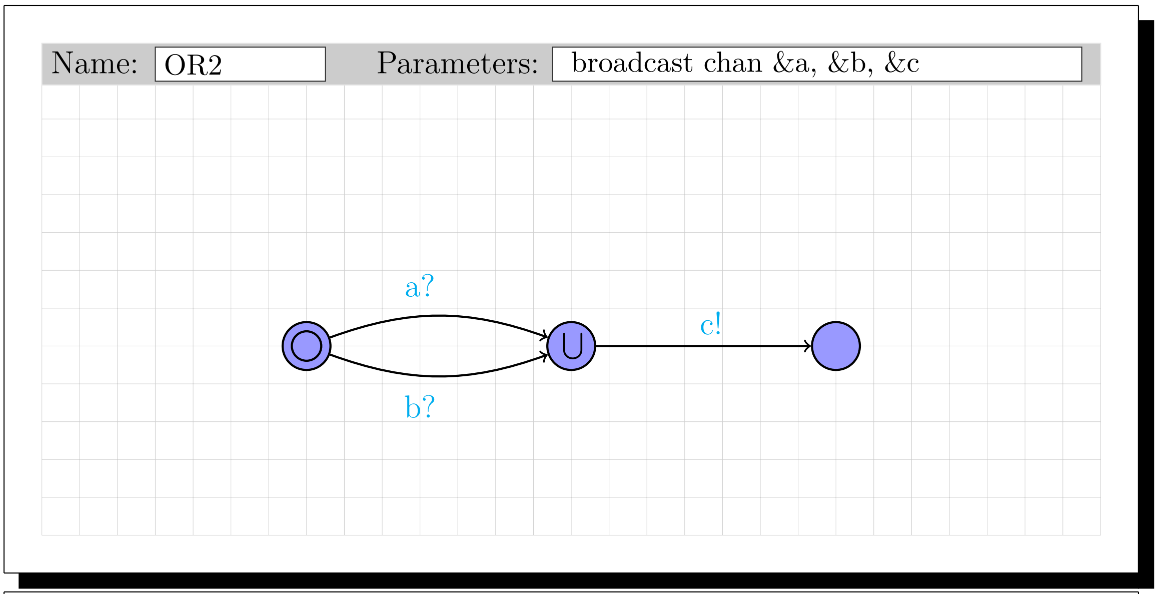
```
// This model is generated by UATMM2Uppaal M2T Generation
// Date: Tue Mar 01 17:37:03 CET 2016

// Broadcast channels.
broadcast chan EnterRoom_success, Window_success, BreakGlass_success, Climb_success,
    Door_success, UseKey_success, OpenDoor_success, Wall_success, Explosive_success,
    Hammer_success;

// Global Booleans
bool f = false;
bool t = true;
bool system_done = false;

bool result_SatScenario3 = true;

bool SatScenario3[6] = {true, false, true, false, true, true}

void handleSatScenario3(int value) {
        result_SatScenario3 = result_SatScenario3 and SatScenario3[value];
}
```

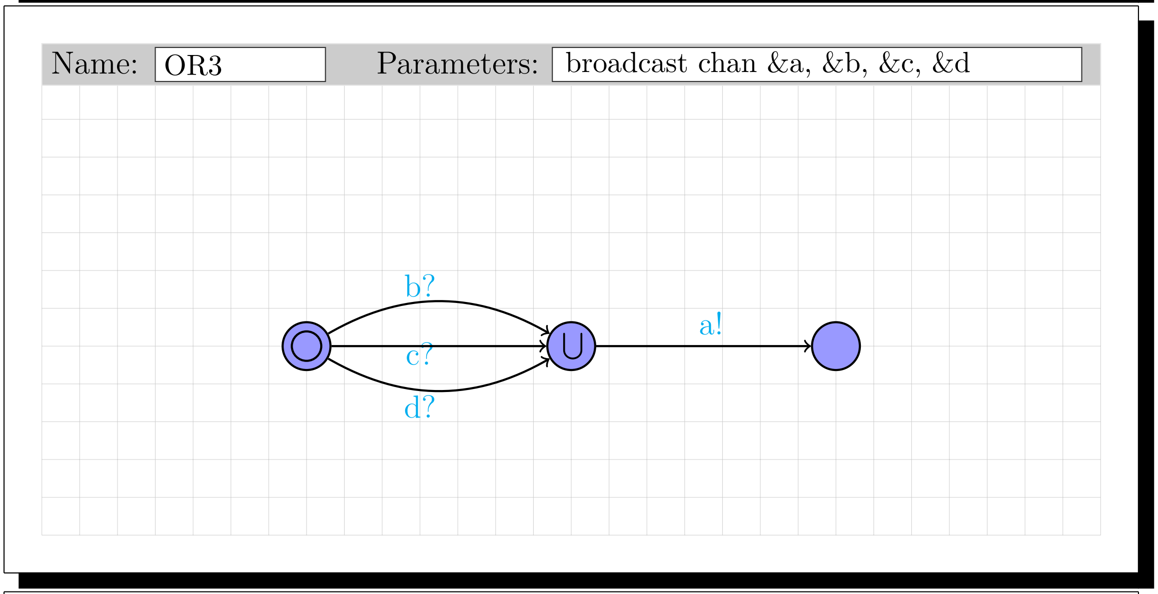
Generated code within the system part

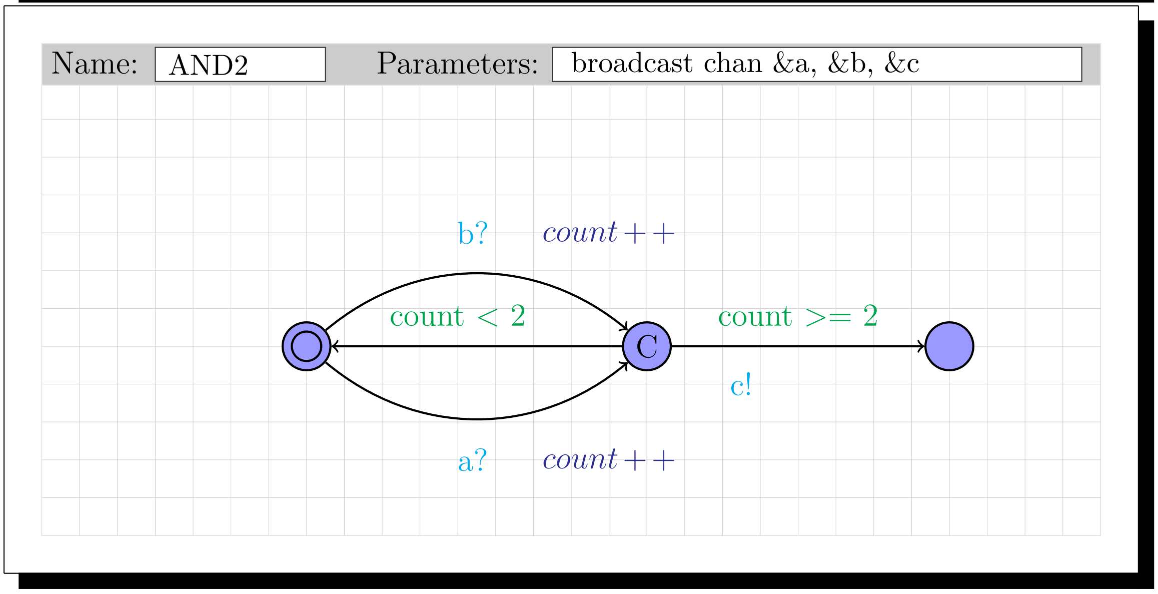
Code 6.2: Code generated for the system part

```
// This model is generated by UATMM2Uppaal M2T Generation
// Date: Tue Mar 01 17:37:03 CET 2016

// Basic Elements
d_BreakGlass = TimedBE(BreakGlass_success, 2, 3, 0);
d_Climb = TimedBE(Climb_success, 9, 15, 1);
d_UseKey = TimedBE(UseKey_success, 1, 2, 2;
d_OpenDoor = TimedCBE(OpenDoor_success, UseKey_success, 1, 2, 3);
d_Explosive = TimedBE(Explosive_success, 3, 8, 4);
d_Hammer = TimedBE(Hammer_success, 25, 50, 5);

// AND gates
d_Window = AND2(Window_success, BreakGlass_success, Climb_success);
d_Door = AND2(Door_success, UseKey_success, OpenDoor_success);

// OR gates
d_EnterRoom = OR3(EnterRoom_success, Window_success, Door_success, Wall_success);
d_Wall = OR2(Wall_success, Explosive_success, Hammer_success);

Sys = System(EnterRoom_success);

// System
system Sys, d_BreakGlass, d_Climb, d_UseKey, d_OpenDoor, d_Explosive,
        d_Hammer, d_Window, d_Door, d_EnterRoom, d_Wall;
```

Templates for the Gates







Chapter 7

Results

This thesis achieved the following results.

1. *Multi-parameter Attack Tree model in Uppaal.* Based upon the state of the art, this thesis has delivered a set of Uppaal-template and the correct logic to combine them in a fully functional Uppaal model, which represents the behavior of an *Attack Tree (AT)*. Major advantages above existing tools are the ability to analyze across multiple domains, the use of sequential behavior and the possiblity to properly use shared subtrees. All these advantages come with using Timed Automata and the versatility of *Uppaal.* The versatility of *Uppaal* mainly expresses itself in the multiple extensions, which have different focuses during its analysis, but with the exception of a few small details have a combined underlying structure. The predictions are that more of these features will be integrated in the general tool, as has been done for the *Statistical Model Checker* (Uppaal SMC), which has been fully integrated in Uppaal since version 4.14.

   The set of templates that we created to represent a solid Uppaal Model and used during the transformation process can be found in Appendix E.

2. *Attack Tree MetaModel (ATMM)*, abstracting the multiple tools into a more generic intermediate model allows future extensions to easily adapt and use the

already existing transformations.For this we also present a meta-model, named *Attack Tree Meta Model (ATMM)* can in be found in Appendix F. For a more in detail explanation of this model, we refer the reader to Chapter 3. Since we took the current state of the art into consideration, the meta-model can be used for most of the Attack Tree formalisms that currently exist, as long as these formalisms conform to the basic tree structures. All non-tree like constructions are not included.

3. *Transformations.* Multiple transformations are included. For more details on the individual transformations we refer the reader to Chapter 4.

   (a) *ADTool2ATMM.etl.* A transformation in ETL from the *ADTool* xml output into an instance of the *ATMM*, its source code can be found in Appendix A.

   (b) *ATMM2ADTool.etl.* Inverse of the previous transformation, also using ETL. The actual transformation can be found in Appendix B.

   (c) *ATMM2Uppaal.egl.* Using EGL to transform an instance of the meta-model into a working Uppaal-model. The initial ATMM2Uppaal.egl can be found in Appendix C and for an example of the imported EGL files that are used within this template, we refer the reader to Appendix D.

   (d) *UppaalEcore2Uppaal.egl.* Separate from the ATMM transformations, the need for additional transformation arose. This transformation uses an *Uppaal Meta-Model* developed by XX University and transforms it into an actual XML model.

4. *Tool* incorporating the transformations, one generic tool which can be loaded with the different transformations. A maven build which streamlines the two transformations and provides a jar file which can be used as a command line tool to transform ADTool xml into a Uppaal model. The code can be found in Appendix G.

   (a) Version 1, uses *ADTool2ATMM* and *ATMM2Uppaal*, this delivers a chain of transformations which takes ADTool as an input and an Uppaal-model as its output.

   (b) Version 2, only has the *UppaalEcore2Uppaal.* Takes an instance of the meta-model as input and produces the Uppaal-model.

7.1 Discussion

When the original plans for this thesis were drafted, the goal was to generate Uppaal models from Attack Trees prefereably in an organized, logical and automated way. Initial designs and idea focussed around creating a tool that could transform inputs from AT tools into an Uppaal model in a rather ad-hoc method. However after these proposals we soon came to realise that this did not fulfill the addititional requirement of automating this from existing tools, and the focuss shifted to a possible extenstion within an existing tool to make the Uppaal model generation more streamlined. After careful consideration we chose to use *ADTool* as the input tool. Primary motivations for this were:

- **Usability**. This tool provides an easy to use *Graphical User Interface (GUI)* which allows for the quick generations of Attack Trees.
- **Existing Export-to-XML features**. This export feature gives a XML format representing this AT, this structure could then be easily used as input for the *ATMM*.
- **Availability of the code**. A major advantage was the source code, which was publicly available, very clearly structered and could easily be extended by the Export-to-Uppaal Module.
- **Usage within the chair**. The fact that it is also used in the FMT chair cannot be ignored as a major factor in this design process, both from a usability point of view (i.e. the focus of this thesis should be on something that is actually going to be used) and from a expertise point of view (i.e. Having experts available for questions surrouding how it is actually used is very useful).

This process was the result of my research topics (a pre-thesis requirement) and with this goal in mind (*Extending ADTool with an Export-to-Uppaal module*) I started this thesis-project. However when it finally came down to proposing and demoing this solution to other people on staff some things changed. Some meetings and brainstorm sessions changed the initial requirements to an intermediate tool which could transform output of ADTool into input for Uppaal, almost as we initially approached this problem, but more structured and more ad-hoc. Arend Rensink introduced model transformations into this process and this is were the first ideas for the final result come from. The possibilities of model transformations languages were explored and eventually we chose for the Epsilon framework and primarly for the Epsilon Transformation Language

(ETL). This language allowed for clean, rule-based transformations and gave us the possibility to implement transformations from different tools into an intermediate model (meta-model).

7.1.1 Evaluation of the possible tools

Looking at the import possibilities there arose a need for a *Graphical User Interface* for the creation of ATs. Since creating one from scratch was beyond the scope of this thesis, the choice was made to pick on of the tools currently available. The primary selection was made by looking at the available tools and selecting the ones that complied to our most important selection criteria, i.e. *scientific foundations*, *currently being maintained* and *available research done with the tool.* This resulted in *SecurITree*, *AttackTree+* and *ADTool.*

SecurITree

Developed by Amenaza, *SecurITree* is a tool to model risks in an organization. It uses ATs to understand how parts of an organization are vulnerable for attacks and can be attacked by an attacker, what harm that attack can do, and what could be used as a countermeasure for these specific attacks [24][46].

AttackTree+

Another commercial software for attack tree modeling, allows the user to model probabilities that an attack will succeed. The application introduces attributes in the form of indicators, which can be used for representing costs for an attack or difficulty [3]. The questions that can be answered with this solution are of the following nature. *What is the fastest possible attack within a cost of x?*.

ADTool

ADTool is open source software used for the graphical modeling and quantitative analysis of *attack–defense trees (ADTrees)* [30]. The tool is developed by the University of Luxembourg and according to Kordy et al.. [30] the following are the beneficial features:

- The tool is a free and open source application.
- It supports quantitative and qualitative analysis of *Attack Trees.*

- The ATs are based on well-founded formal framework.
- *ADTool* guides the user in constructing syntactically correct models.
- Automates the computation of security related parameters.

Comparison

Tool	SecurITree	AttackTree+	ADTool
GUI	yes	yes	yes
Cost	$1,250 / month	free trail	free
Code available	no	no	yes

Table 7.1: Tool Comparison

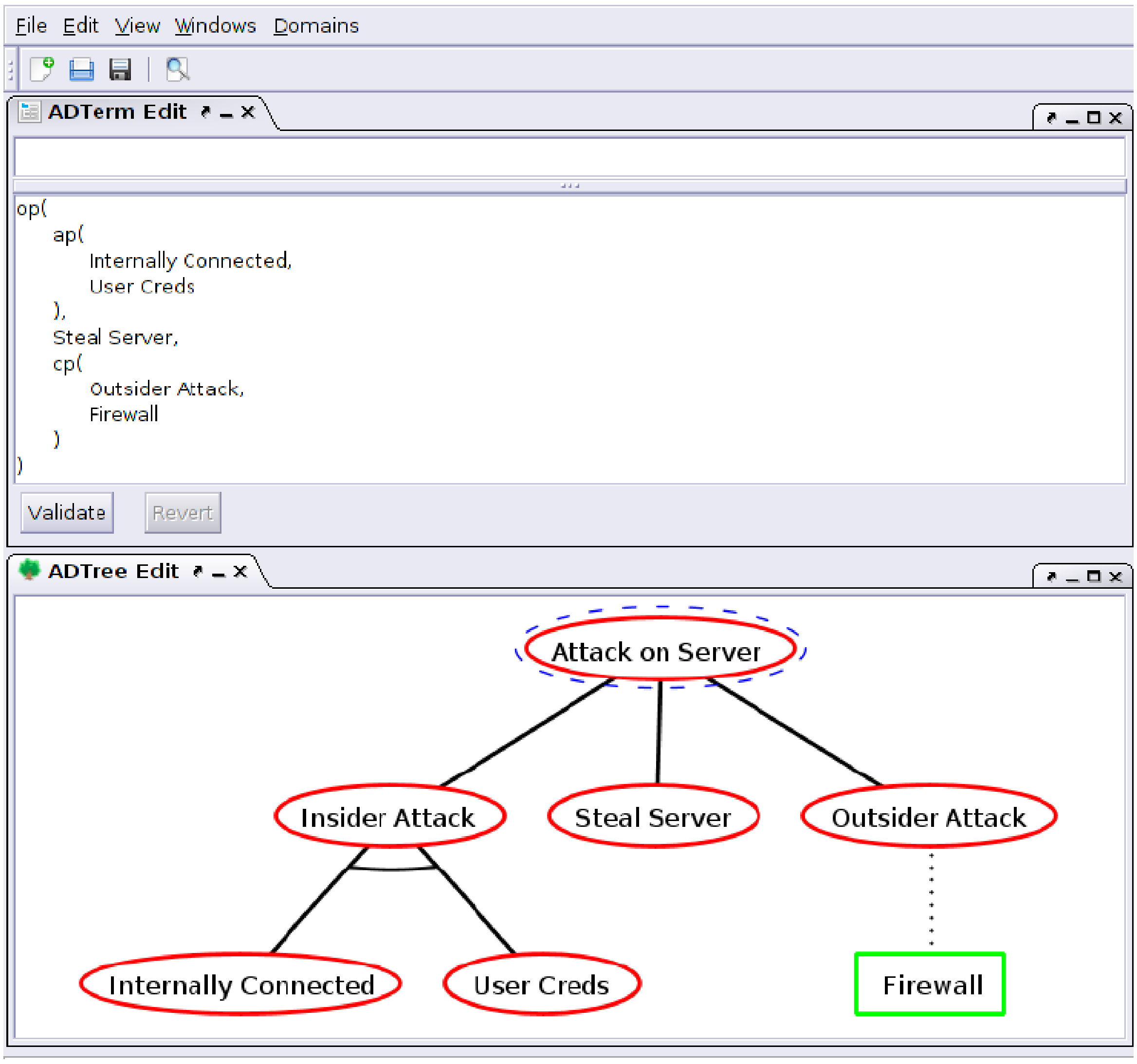
Based on this information, the choice for *ADTool* has been made. It was the cheapest available academic tool that met our criteria and the tool was already in use within our faculty. Its Graphical User Interface is very open and allows for the quick generation of *Attack Trees*, an example of the interface can be found in Figure 7.1. It had an open-source code base, which allowed for easy integration of the tool for the changes we initially attempted. Even though in the final design the choice of input tool was not of the most important, the XML output of *ADTool* allowed for an easy transformation into our *ATMM*.

7.1.2 Evaluation of the model transformation frameworks

For our choice in model transformation languages we looked at research on three prominent representatives, namely ATL, ETL and TGGs.

ATL (ATLAS Transformation Language) is a transformation language originally created by the ATLAS group and can be used to transform sets of input models into sets of output models.

ETL (Epsilon Transformation Language) is part of the Epsilon transformation language and is currently developed by the University of York, has an active forum and nice tutorials.**TGGs** (Triple Graph Grammars), which use the QVT approach of model transformations, is widely used, but lacks large scale implementations because it disregards means for structuring specifications. This makes larger scale transformations unmaintainable.



```
op(
    ap(
        Internally Connected,
        User Creds
    ),
    Steal Server,
    cp(
        Outsider Attack,
        Firewall
    )
)
```

Figure 7.1: A screenshot of the ADTool GUI

Straight off the bat, ATL and ETL have a huge advantage over TGGs, because they strongly implement Ecore models into there transformations. And since the intermediate meta-model we use is an Ecore model, this eliminated the TGGs options. The differences between ETL and ATL are less clear, syntactical they look very similar and the rule based approach used for MDE is using the same concepts. ATL is a m2m component and more mature, this will have benefits regarding remaining bugs in the tooling and useful examples for features. Both have forums, but the ETL one is slightly more active and more rich in examples. ETL is more imperative, allowing for more statements / queries within parent statements. And finally ETL allows for the smooth integration with existing java libraries.

Comparison

	TGGs	ETL	ATL
Ecore supports	no	yes	yes
Rule-based structure	can be	yes	yes
Mature	yes	no	yes
Active forum	depends	yes	yes
Imperative edge		yes	
Java integration	no	yes	no

Table 7.2: Transformation Languages Comparison

This resulted in the choice for ETL, which is part of the **Epsilon Modelling Family** and also gave us access to the closely related EGL (Epsilon Generation Language), which turned out the be

7.2 Recommendation for Future Work

Even though the basis for the framework works as expected, it can be further extended in multiple ways:

Improving current Uppaal model

Different approaches use different templates to represent gates in Uppaal. For examples representing AND gates could be done by counting the number of incoming broadcasting channels or create a template-state for every a actual state. Similarly representing SAND gates can be done by triggering the next inactive *Basic Element* or create a chain of triggers which will eventually trigger the same *Basic Element*. All these different approaches have their own advantages and disadvantages, depending on how the model will be used. e.g. Counting AND gates greatly decrease the amount of possible states with respect to the above mentioned alternative and that would be preferable for larger state spaces. More of these optimizations exist and could be implemented as improvements in future work.

Extending current Uppaal model

Instead of implementing new templates for existing gates another area for future work is implementing new gates. As an example the SOR gates, which is the SAND equivalent of the OR gates.

- This is would require to recreate this as an Uppaal template.
- Add the template to the EGL transformation.
- Add an instance of the template into the system declaration.

Implementing this and other gates into the transformation is something that would really increase the benefits of this work.

Adding additional domains

On top of the existing domains additional attributes can be used in Attack Trees. One example would be to enrich the attributes by making them dependant on each other. This is done by Kumar et al. [34], there they express attributes as components like *cost*, *time* and *skill* and make these components dependent, e.g. time can depend on skill level. This and more extensions are very valuable extensions and should definitely be investigated in future work.

Extending the Attack Tree Meta-Model (ATMM)

This is a process that has already started, the initial design is being extended for its use in the TREsPASS project. Since multiple partners will be using this model, there will be some changes in upcoming versions to improve the meta-model and adaptations of other users.

Increase input for ATMM

This work only has ADTool as an input model for the meta-model, separate work on our university has been done to allow for more information from different sources, primarily from ATCalc and from the Attack Tree standard in TREsPASS. If the meta-model would become a standard to represent ATs, a lot of import methods should be available. This is another direction for future work.

Handling Uppaal output

The next step that needs to be taken in the automated process is porting the results from Uppaal back into a second meta-model. This is one of major next challenges and starts with asking the right questions in Uppaal and extracting the correct answer out of the resulting traces.

Chapter 8

Conclusion

Our goal was to provide an easy way to transform *Attack Trees* into Uppaal models. In the first part of this thesis we identified the key features that need to be represented in ATs and thus be present in the Uppaal model. These features included the following intermediate nodes: a conjunctive gate (*AND*), a disjunctive gate (*OR*) and a sequential conjunctive gate (*SAND*). Furthermore the concept of time needs to be included, leaves have a certain trigger to get them activated and using Uppaal allowed for including the concept of time. Within a timed AT, leaves will have a minimum (x) and a maximum (y) attribute which will make Uppaal activate the represented leaf between x and y time units. Last, looking at the bigger examples of ATs the need arose to allow the sharing of sub-trees to greatly decrease the trees complexity.

Once both the input ATs and the outgoing Uppaal models were defined, the focus of this thesis shifted to supplying an automated process to streamline this transformation. After some careful consideration and the input of domain experts, we decided on a transformation through meta-modelling. This resulted in the second contribution from this thesis, the *Attack Tree Meta Model (ATMM)*. This meta-model is used as an intermediate standard to represent a majority of all the Attack Tree formalisms currently used in the scientific community.

Finally, for completion we implement the actual transformations. Firstly from our input model (*ADTool*) into our meta-model (*ATMM*) and vice versa. And secondly from our meta-model (*ATMM*) into our output model (*Uppaal*).

Overall this thesis proved that the goals set within the preliminarily research have been completed into a clear, fast and reliable command line tool. Which transforms the XML export from ADTool into an input file for *Uppaal*. It furthermore showed how quickly this transformation process could be changed to implement a different type of Uppaal output which makes it much more valuable for future research.

The results show the possibilities of model transformations in the context of tree based formalisms. These primary results are so promising that currently a lot of work within the faculty is taking a similar redirection into the field of model-transformations.

As a recommendation for future research, we strongly recommend two separate directions. Firstly in implementing the results of this thesis in a larger tool chain surrounding Attack Tree analysis. The start for this has already been initialized. Future research will include the results of questions on the generated Uppaal model and porting these results back into the AT formalism. Secondly we recommend to keep improving the provided transformations, primarily the resulting Uppaal model. This is subject to change once the chain is completed and there is a need to ask additional questions on the Uppaal model.

All in all, this was a great learning experience and the final result proved that using model transformations for tree-based formalisms proved beneficial and offered new insights for future research.

Bibliography

[1] ADTool Universite du Luxembourg. http://satoss.uni.lu/members/piotr/adtool/. Accessed: 2014-08-15.

[2] LIST Web Sandbox – ATCalc. http://fmt.ewi.utwente.nl/puptol/atcalc/. Accessed: 2015-12-15.

[3] Isograph: AttackTree+. http://www.isograph.com/software/attacktree/. Accessed: 2014-12-23.

[4] The TREsPASS Project (Technology-supported Risk Estimation by Predictive Assessment of Socio-technical Security). http://www.trespass-project.eu/. Accessed: 2015-09-30.

[5] UPPAAL. http://www.uppaal.org. Accessed: 2014-06-15.

[6] UPPAAL for Planning and Scheduling. http://people.cs.aau.dk/~adavid/cora/. Accessed: 2015-09-30.

[7] Kronos: A model-checking tool for real-time systems. volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer Berlin / Heidelberg, 1998.

[8] F. Arnold, H. Hermanns, R. Pulungan, and M. I. A. Stoelinga. Time-dependent analysis of attacks. In *Third International Conference on Principles and Security of Trust, Grenoble, France*, volume 8414, pages 285–305, Berlin, April 2014.

[9] Schneier B. Attack Trees - Modeling security threats. *Dr. Dobb's Journal*, December 1999.

[10] D. Baca and K. Petersen. Prioritizing countermeasures through the countermeasure method for software security (cm-sec). In *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, pages 176–190. 2010.

[11] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[12] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[13] S. Bistarelli, F. Fioravanti, and P. Peretti. Defense trees for economic evaluation of security investments. In *Availability, Reliability and Security, 2006. ARES 2006.*, April 2006.

[14] Johannes A. Buchmann. *Introduction to Cryptography.* Springer, 2 edition, 2004. p. 190.

[15] E. Bursztein, B. Benko, D. Margolis, T. Pietraszek, A. Archer, A. Aquino, A. Pitsillidis, and S. Savage. Handcrafted fraud and extortion: Manual account hijacking in the wild. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 347–358, New York, NY, USA, 2014. ACM.

[16] S.A. Camtepe and B. Yener. Modeling and detection of complex attacks. In *Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. Third International Conference on*, pages 234–243, Sept 2007.

[17] A. David, K.G. Larsen, A. Legay, M. Mikučionis, D.B. Poulsen, J. Van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'11, pages 80–96, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Gurpreet Dhillon and Steve Moore. Computer crimes: theorizing about the enemy within. *Computers & Security*, 20(8):715–723, 2001.

[19] K.S. Edge, G.C. Dalton, R.A. Raines, and R.F. Mills. Using attack and protection trees to analyze threats and defenses to homeland security. In *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pages 1–7, Oct 2006.

[20] K.S. Edge, R.A. Raines, M.R. Grimaila, R.O. Baldwin, R.W. Bennington, and C.E. Reuter. The use of attack and protection trees to analyze security for an online banking system. In *HICSS*, page 144. IEEE Computer Society, 2007.

[21] I.N. Fovino, M. Masera, and A. De Cian. Integrating cyber attacks within fault trees. *Reliability Engineering & System Safety*, pages 1394 – 1402, 2009. the 18th European Safety and Reliability Conference.

[22] T.A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463. Springer, 1997.

[23] C. Herley. Why do nigerian scammers say they are from nigeria? *WEIS*, 2012.

[24] T.R. Ingoldsby. Attack tree-based threat risk analysis. *Amenaza Technologies Ltd. Copyright*, 2010, 2009.

[25] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Sasa Radomirovic, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction.

[26] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. The epsilon book. *Structure*, 178:1–10, 2010.

[27] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062, page 200, 2006.

[28] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. The epsilon transformation language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] B. Kordy, S Mauw, S. Radomirović, and P. Schweitzer. Foundations of attack-defense trees. In *Proceedings of the 7th International Conference on Formal Aspects of Security and Trust*, pages 80–95, 2011.

[30] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer. Adtool: Security analysis with attack-defense trees (extended version). *CoRR*, abs/1305.6829, 2013.

[31] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer. Adtool: Security analysis with attack-defense trees. In *10th International Conference on Quantitative Evaluation of Systems (QEST), Buenos Aires, Argentina*, volume 8054 of *Lecture Notes in Computer Science*, pages 173–176. Springer, August 2013.

[32] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 2014.

[33] N. Kshetri. The simple economics of cybercrimes. *IEEE Security and Privacy*, Vol. 4, No. 1, January/February 2006.

[34] R. Kumar, E. J. J. Ruijters, and M. I. A. Stoelinga. Quantitative attack tree analysis via priced timed automata. In S. Sankaranarayanan and E. Vicario, editors, *Proceedings of the 13th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2015), Madrid, Spain*, volume 9268 of *Lecture Notes in Computer Science*, pages 156–171, September 2015.

[35] W. Lv and W. Li. Space based information system security risk evaluation based on improved attack trees. In *Third International Conference on Multimedia Information Networking and Security*, MINES '11, pages 480–483, 2011.

[36] I. S. Moskowitz and M. H. Kang. An insecurity flow model. In *Proc. Workshop on New Security Paradigms, Langdale, Cumbria, United Kingdom*, 1997.

[37] S.C. Patel, J.H. Graham, and P.A.S. Ralston. Quantitatively assessing the vulnerability of critical information systems: A new method for evaluating security enhancements. *International Journal of Information Management*, 28(6):483 – 491, 2008.

[38] L.M. Rose, R.F. Paige, D.S. Kolovos, and F. Polack. The epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[39] A. Roy, D.S. Kim, and K.S. Trivedi. Act: Attack countermeasure trees for information assurance analysis. In *IEEE Conference on Computer Communications Workshops , 2010*, March 2010.

[40] A. Roy, D.S. Kim, and K.S. Trivedi. Cyber security analysis using attack countermeasure trees. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 28:1–28:4, New York, NY, USA, 2010. ACM.

[41] M. Sanford, D. Woodraska, and D. Xu. Security analysis of filezilla server using threat models. *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 678–682, 2011.

[42] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[43] C.-W. Ten, G. Manimaran, and C.-C. Liu. Cybersecurity for critical infrastructures: Attack and defense modeling. *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, 40(4):853–865, 2010.

[44] K. S. Trivedi, D. S. Kim, A. Roy, and D. Medhi. Dependability and security models. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, pages 11–20, Oct 2009.

[45] J. D. Weiss. A system security engineering process. *14th National Computer Security Conference*, pages 572–581, 1991.

[46] S.M. Welberg. Vulnerability management tools for cots software - a comparison, February 2008.

[47] R. Yager. OWA trees and their role in security modeling using attack trees. *Information Sciences*, pages 2933 – 2959, 2006.

```
}
```

Chapter 9
Appendices

9.1 Appendix A: ADTool2ATMM (ETL)

```
pre {
        var javaDate = new Native("java.util.Date");
        javaDate.toString().println("Starting ADTool2M
           transformation: ");

        // Start a recursive XML Teardown to determine
           parent/children relations.
        // (Operation instead of rules, as rules did not seem
           suitable)
        //ADTool.root.e_node.recursiveXMLTeardown(true, null);
}

post {
        var javaDate = new Native("java.util.Date");
        javaDate.toString().println("Completed ADTool2M
           transformation: ");
}
```

```
// Transform all domains and set several attributes by
   obtaining
// info from attributes and child nodes
@primary
rule ADToolDomain2UATDomain
        transform ad : ADTool!t_domain
        to gad : UATMM!Domain {

        gad.ID = ad.a_id;
        gad.Computation = true;
}

rule ADToolTree2UATTree
        transform ad : ADTool!t_adtree
        to uat : UATMM!AttackTree {

        var role1 = new UATMM!Role();
        role1.RoleType = UATMM!RoleType#Contributing;
        role1.RoleDescription = "Attack";

        var role2 = new UATMM!Role();
        role2.RoleType = UATMM!RoleType#Counteracting;
        role2.RoleDescription = "Defense";

        uat.Roles.add(role1);
        uat.Roles.add(role2);

        for(domain in UATMM!Domain.allInstances){
                uat.Domains.add(domain);
        }

        ad.e_node.recursiveXMLTeardown(true, null);

}

// Recursive XML Teardown operation. Called on XML node and
   will call method on all child nodes
```

```
// Creates an ATNode, sets its parameters and sets
   parent/child relation to child ATNodes.
operation ADTool!t_node recursiveXMLTeardown(role : Boolean,
   parent : UATMM!Node) : UATMM!Node{
        var children = self.c_node;
        var parameters = self.c_parameter;
        var result = new UATMM!Node();

        result.Label = self.e_label.text;

        // Reverse roles if node contains switchRole attribute
        if(("yes").equals(self.a_switchRole)){
                role = not role;
        }

        result.Role = self.findRole(role);
        result.Connector = self.findConnector();

        getAttackTree().Nodes.add(result);

        // If there is a parent, add it.
        if(parent<>null){
                result.Parents.add(parent);
        } else {
                getAttackTree().Root = result;
        }

        // If node has any childeren, call method recursively
           and add references to results.
        if(children.size()>0){
                for(c in children){
                        result.Children.add(c.recursiveXMLTeardown(role,
                           result));
                }

        // If there are not any childeren, there can be
           paramters. Add
        } else {
                for(p in parameters){
                        var param = new UATMM!Attribute();


                        param.Value = p.text;
                        param.Domain = p.findDomain();
                        result.Attributes.add(param);
                }
        }
        return result;
}

operation ADTool!t_parameter findDomain(){
        return UATMM!Domain.allInstances
           .select(d|d.ID==self.a_domainId).first();
}

operation ADTool!t_node findConnector(){
        if(self.a_refinement.equals("disjunctive")){
                return new UATMM!OR();
        } else {
                return new UATMM!AND();
        }
}

operation ADTool!t_node findRole(role:Boolean){
        if(role){
                return UATMM!Role.allInstances
                   .select(n|n.RoleType=
                   UATMM!RoleType#Contributing).first();
        } else {
                return UATMM!Role.allInstances
                   .select(n|n.RoleType=
                   UATMM!RoleType#Counteracting).first();
        }

}

@cached
operation Any getAttackTree() : UATMM!AttackTree{
        return UATMM!AttackTree.allInstances.last();
}
```

9.2 Appendix B: ATMM2ADTool (ETL)

```
pre {
        var javaDate = new Native("java.util.Date");
        javaDate.toString().println("Starting ADTool2M
           transformation: ");
}

post {
        var javaDate = new Native("java.util.Date");
        javaDate.toString().println("Completed ADTool2M
           transformation: ");
}

@primary
rule UATMM2ADTool
        transform at : UATMM!AttackTree
        to adt : ADTool!t_adtree {
        ADTool.root = adt;
}

rule GAT2ADTool
        transform r : UATMM!Node
        to f : ADTool!t_node {

        //Set refinement attribute of node
        f.a_refinement= r.findRefinement();

        // Create and add label to the current node.
        var label = new ADTool!t_label;
        label.text = r.Label;
        f.appendChild(label);

        // Append all attribute nodes to the current node
        // Invoke the lazy rule for all attributes (by using
           equivalent())
        // then append all children XML nodes to the current
           node.
        for (child in r.Attributes.equivalent()) {
        f.appendChild(child);
        }

        // If the parent ATNode has a different role, add
           'switchRole' attribute to the XML node.
        if(r.Parents.size() > 0 and
           r.Parents.first().Role.RoleType <> r.Role.RoleType
           ){
                f.a_switchRole = "yes";
        }

        // Append all child ATNodes to the current node in the
           XML
        for (child in r.Children.equivalent()) {
                f.appendChild(child);
        }

        // Append the current node to the root (Not sure why
           this works, as it is called for all nodes)
        ADTool.root.appendChild(f);
}

// Simply transform each domain into an XML domain and attach
   it to the root
rule Domain2ADToolDomain
        transform at : UATMM!Domain
        to ad : ADTool!t_domain {

        var class = new ADTool!t_class;
        class.text = at.determineClass();
        var tool = new ADTool!t_tool;
        tool.text = "ADTool";
        ad.a_id = at.ID;

        ad.appendChild(class);
        ad.appendChild(tool);
        ADTool.root.appendChild(ad);
}
```

```
@lazy
rule Attribute2Param
        transform a : UATMM!Attribute
        to p : ADTool!t_parameter{

        p.a_domainId = a.Domain.ID;
        p.text = a.Value;
}

// Find the string version of the refinement of ATNode
operation UATMM!Node findRefinement() : String{
        if(self.Connector.isTypeOf(UATMM!OR)){
                return "disjunctive";
        } else if(self.Connector.isTypeOf(UATMM!AND)){
                return "conjunctive";
        }
        return "disjunctive";
}

// Find the string version of the refinement of ATNode
operation UATMM!Domain determineClass() : String{
        //Removing trailing digits to determine class
        return "lu.uni.adtool.domains.predefined."+
           (self.ID.replaceAll("\\d*$", ""));
}
```

9.3 Appendix C: ATMM2Uppaal (EGL)

```
[%
        var location = new
           Native("com.utwente.standalone.Locator");
        TemplateFactory.setTemplateRoot(location.getURI()+"predefined/");

        var andTemplate : Template :=
           TemplateFactory.load('ANDTemplate.egl');
        var orTemplate : Template :=
           TemplateFactory.load('ORTemplate.egl');
        var systemTemplate : Template :=
           TemplateFactory.load('SystemTemplate.egl');
        var generated : Template :=
           TemplateFactory.load('generated.egl');
        var timedBETemplate : Template :=
           TemplateFactory.load('TimedBETemplate.egl');
        var expBETemplate : Template :=
           TemplateFactory.load('ExpBETemplate.egl');
        var timedCBETemplate : Template :=
           TemplateFactory.load('TimedCBETemplate.egl');
        var expCBETemplate : Template :=
           TemplateFactory.load('ExpCBETemplate.egl');
        var ElementId2LeafId : Map;
        var LeafId2ElementId : Map;
        var id = 0;
        for(n in AT!Node.allInstances().select(n|
           n.Children.size() == 0)) {
                ElementId2LeafId.put(n.Label, id);
                LeafId2ElementId.put(id, n.Label);
                id = id + 1;
        }
        var domain2type = initDomainType();
        var domain2initValue = initDomainInitValue();
        var domain2metaDomain : Map;
        var channelsUsed : Sequence;
        var namesUsed : Sequence;
        var andTemplatesIncluded : Sequence;
        var orTemplatesIncluded : Sequence;
%]<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN'
   'http://www.it.uu.se/research/group/darts/uppaal/flat-1_2.dtd'>
<nta>
        <declaration>
[%=generated.process()%]
// Broadcast channels.
broadcast chan [% var b : Boolean = false;
for( n in AT!Node.allInstances.select(n |
   (n.Role.RoleDescription == "Attack" or n.Children.size() ==
   0))) {
if (channelsUsed.indexOf(n.getConvertedLabel()) == -1) {
        channelsUsed.add(n.getConvertedLabel());
        if(b){%], [% } else {b = true; }
        %][%=n.getConvertedLabel() %]_success[% }} %];
// Global Booleans
bool f = false;
bool t = true;
bool system_done = false;
bool system_busy = false;
[%
        var domains : Sequence;
        var domains_stripped : Sequence;
        for(d in AT!Domain.allInstances()) {
                domains.add(d.ID);

                var id_stripped = d.ID.replaceAll("\\d","");
                domain2metaDomain.put(id, id_stripped);
                if(domains_stripped.indexOf(id_stripped) ==
                   -1) {
                        domains_stripped.add(id_stripped);
                }
        }
        // Timed or Distribution based
        var timed = ((domains_stripped.indexOf("MinTimeSeq")
           <> -1) or (domains_stripped.indexOf("MinTimePar")
           <> -1 ));
        var timed_domains : Sequence;
        for ( d in domains ) {
                if(d.startsWith("MinTimePar") or
                   d.startsWith("MinTimeSeq")) {
                        timed_domains.add(d);
                }
        }
        (">> # of timed domains:
           "+timed_domains.size()).println();
        var min_domain;
        var max_domain;
        if (timed_domains.size() == 2) {
                var first_value;
                var second_value;
                // get the all the attributes of the first
                   leaf, based on this decide which domain is
                   min / max
                for ( att in
                   getATNode(LeafId2ElementId.get(LeafId2ElementId
                                .keySet().first())).Attributes){
                        if
                           (timed_domains.indexOf(att.Domain.ID)
                           == 0) {
                                first_value =
                                   att.Value.asReal().floor();
                        }
                        if
                           (timed_domains.indexOf(att.Domain.ID)
                           == 1) {
                                second_value =
                                   att.Value.asReal().floor();
                        }
                }
                if (first_value < second_value) {
                        min_domain = timed_domains.at(0);
                        max_domain = timed_domains.at(1);
                } else {
                        min_domain = timed_domains.at(1);
                        max_domain = timed_domains.at(0);
                }
        } else if (timed_domains.size() == 1) {
                        min_domain = timed_domains.at(0);
                        max_domain = timed_domains.at(0);
        }
        if ((domains_stripped.indexOf("DiffLMH") <> -1) or
           (domains_stripped.indexOf("DiffLMHE") <> -1 )){
                %] // Values for LMHE domain:"
const int L = 1; // Low"
const int M = 2; // Medium
const int H = 3; // High
const int E = 4; // Extreme
                [%
        }
        for ( d in domains ) {

                var str_d = d.replaceAll("\\d","");

                if (not domain2type.containsKey(str_d) )
                   ("Couldn't match domain ["+d+"]").println;
                else {
                        %][%=domain2type.get(str_d) %]
                           result_[%=d %] =
                           [%=domain2initValue.get(str_d) %];

                        [%
                }
        }
        for ( d in domains ) {
                var str_d = d.replaceAll("\\d","");
                if (not domain2type.containsKey(str_d) )
                   ("Couldn't match domain
                   ["+str_d+"]").println;
                else if (d.startsWith("MinTime")) {
                } else {
                        %][%=domain2type.get(str_d) %] [%=d
                           %][[%=LeafId2ElementId.size() %]] =
                           {[%

                        var first = true;
for( key in LeafId2ElementId.keySet() ) {
        for ( att in
           getATNode(LeafId2ElementId.get(key)).Attributes){
                if (first) {
                if (att.Domain.ID.equals(d)) {
                        first = false;
                        if  (str_d.equals("ProbSucc")) {
                                %][%=(1000*att.Value).floor()
                                   %][%
                        } else {
                                %][%=(att.Value) %][%
                        }
                }
                } else {
                if (att.Domain.ID.equals(d)) {
                        if  (str_d.equals("ProbSucc")) {
                                var newVar =
                                   ((1000*att.Value).floor());
                                %], [%=newVar %][%
                        } else {
                                %], [%=(att.Value) %][%
                        }
                }
                }
        }
}
                        %]};

                        [%
                }
        }
        var str_d : String;
        for ( d in domains ) {
                if (d.startsWith("DiffLMH")) {
                        %]
void handle[%=d %](int value) {
        if ([%=d %][value] > result_[%=d %]) {
                result_[%=d %] = [%=d %][value];
         }
```

```
}
                [%
                } else if  (d.startsWith("MinCost") or
                   d.startsWith("MinSkill")){
                %]
void handle[%=d %](int value) {
        result_[%=d %] = result_[%=d %] + [%=d %][value];
}
                [%
                } else if  (d.startsWith("SatScenario") or
                   d.startsWith("SatProp")){
                %]
void handle[%=d %](int value) {
        result_[%=d %] = result_[%=d %] and [%=d %][value];
}
                [%
        } else if  (d.startsWith("ProbSucc")){
                %]
void handle[%=d %](int value) {
        result_[%=d %] = (result_[%=d %] * [%=d
           %][value])/1000;
}
                [%
        }
        }
%]
        </declaration>
[%
var nodes : Sequence;
nodes =  AT!Node.allInstances().select(n| n.Children.size() <>
   0 and (n.Connector.isTypeOf( AT!AND ) or
   n.Connector.isTypeOf( AT!PAND)));
for(n in nodes) {
if (andTemplatesIncluded.indexOf(n.Children.size()) == -1 ) {
        andTemplate.populate('n', n.Children.size());%]
[%=andTemplate.process()%]
[% andTemplatesIncluded.add(n.Children.size());
} } %]
[%
```

```
var nodes : Sequence;
nodes =  AT!Node.allInstances().select(n| n.Children.size() <>
   0 and n.Connector.isTypeOf( AT!OR ));
for(n in nodes) {
if (orTemplatesIncluded.indexOf(n.Children.size()) == -1 ) {
        orTemplate.populate('n', n.Children.size());%]
[%=orTemplate.process()%]
[% orTemplatesIncluded.add(n.Children.size());
} }
if(timed) {
        timedBETemplate.populate('domains', domains);
        %][%=timedBETemplate.process() %][%
        if (AT!Node.allInstances().select(n|
           n.Connector.isTypeOf(AT!PAND)).size() <> 0) {
                timedCBETemplate.populate('domains', domains);
                %][%=timedCBETemplate.process() %][%
        }
} else {
        expBETemplate.populate('domains', domains);
        %][%=expBETemplate.process() %][%
        if (AT!Node.allInstances().select(n|
           n.Connector.isTypeOf(AT!PAND)).size() <> 0) {
                expCBETemplate.populate('domains', domains);
                %][%=expCBETemplate.process() %][%
        }
}
%][%=systemTemplate.process() %]
        <system>
[%=generated.process()%]
// Basic Elements
[% for(n in AT!Node.allInstances().select(n| n.Children.size()
   == 0)) {
if (namesUsed.indexOf(n.getConvertedLabel()) == -1) {
        namesUsed.add(n.getConvertedLabel());
        if(timed) {
                var min = 0;
                var max = 0;
                if (timed_domains.size() == 2) {
                        // Model is timed retreiving domain id
                for ( att in
                   getATNode(n.Label).Attributes){
                        if
                           (att.Domain.ID.equals(min_domain))
                           {
                                min =
                                   att.Value.asReal().floor();
                        } else if
                           (att.Domain.ID.equals(max_domain))
                           {
                                max =
                                   att.Value.asReal().floor();
                        }
                }
        } else if (timed_domains.size() == 1) {
                for ( att in
                   getATNode(n.Label).Attributes){
                        if
                           (att.Domain.ID.equals(min_domain))
                           {
                                min =
                                   att.Value.asReal().floor();
                                max =
                                   att.Value.asReal().floor()
                                   + 1;
                        }
                }
        }
        var leftSibling = n.getLeftSiblingsActivator();
        if (leftSibling.equals("")) {
                %]d_[%=n.getConvertedLabel() %] =
                   TimedBE([%=n.getConvertedLabel()
                   %]_success, [%=min %], [%=max %],
                   [%=ElementId2LeafId.get(n.Label)
                   %]);[%
        } else {
                %]d_[%=n.getConvertedLabel() %] =
                   TimedCBE([%=n.getConvertedLabel()
                   %]_success, [%=leftSibling
```

```
                           %]_success, [%=min %], [%=max %],
                           [%=ElementId2LeafId.get(n.Label)
                           %]);[%
                }
        } else {
                var leftSibling = n.getLeftSiblingsActivator();
                if (leftSibling.equals("")) {
                        %]d_[%=n.getConvertedLabel() %] =
                           ExpBE([%=n.getConvertedLabel()
                           %]_success, 0.5,
                           [%=ElementId2LeafId.get(n.Label)
                           %]);[%
                } else {
                        %]d_[%=n.getConvertedLabel() %] =
                           ExpCBE([%=n.getConvertedLabel()
                           %]_success, [%=leftSibling
                           %]_success, [%=min %], [%=max %],
                           [%=ElementId2LeafId.get(n.Label)
                           %]);[%
                }
        }
} %]
[% } %]
// AND gates
[% for(n in AT!Node.allInstances().select(n| n.Children.size()
   <> 0 and ( n.Connector.isTypeOf( AT!AND ) or
   n.Connector.isTypeOf( AT!PAND ) ) ) ) {
if (namesUsed.indexOf(n.getConvertedLabel()) == -1) {
        namesUsed.add(n.getConvertedLabel()); %]
d_[%=n.getConvertedLabel() %] = AND[%=n.Children.select(c |
   c.Role.RoleDescription == "Attack").size()
   %]([%=n.getConvertedLabel() %]_success[% for(c in
   n.Children.select(c | c.Role.RoleDescription == "Attack"))
   { %], [%=c.getConvertedLabel() %]_success[% } %]);
[% }
} %]
// OR gates
[% for(n in AT!Node.allInstances().select(n| n.Children.size()
   <> 0 and n.Connector.isTypeOf( AT!OR ))) {
```

```
if (namesUsed.indexOf(n.getConvertedLabel()) == -1) {
        namesUsed.add(n.getConvertedLabel()); %]
d_[%=n.getConvertedLabel() %] = OR[%=n.Children.select(c |
   c.Role.RoleDescription == "Attack").size()
   %]([%=n.getConvertedLabel() %]_success[% for(c in
   n.Children.select(c | c.Role.RoleDescription == "Attack"))
   { %], [%=c.getConvertedLabel() %]_success[% } %]);
[% }
} %]
Sys =
   System([%=AT!AttackTree.allInstances().first().Root.getConvertedLabel()
   %]_success);
// System
system Sys[%
for(n in namesUsed) {
%], d_[%=n %][%
}
%];</system>
</nta>
[%
        var javaDateEnd = new Native("java.util.Date");
        System.out.println("> UATMM Model2Text finished on: "+
           javaDateEnd);
        var elapsed = javaDateEnd.getTime() -
           javaDateBegin.getTime();
        var minutes = elapsed / 60000;
        var rest = elapsed.mod(60000);
        var seconds = rest / 1000;
        var mili = rest.mod(1000);
        System.out.print("> Time elapsed: ");
        if (minutes <> 0) System.out.print( minutes + " min,
           ");
        if (seconds <> 0) System.out.print( seconds + " sec :
           ");
        System.out.println( mili+ " ms" );

        operation Any getConvertedLabel() : String {
                return self.Label.replace(" ", "_");
        }
```

```
operation Integer mod(i : Integer) {
        return self - (self/i * i);
}
// Domain Information
operation Any initDomainType() : Map {
        var domain2type : Map;
        domain2type.put("DiffLMH","int");
        domain2type.put("DiffLMHE","int");
        domain2type.put("MinCost","int");
        domain2type.put("MinSkill","int");
        domain2type.put("MinTimePar","double");
        domain2type.put("MinTimeSeq","double");
        domain2type.put("PowerCons","double");
        domain2type.put("ProbSucc","int");
        domain2type.put("ReachPar","int");
        domain2type.put("ReachSeq","int");
        domain2type.put("SatOpp","bool");
        domain2type.put("SatProp","bool");
        domain2type.put("SatScenario","bool");
        return domain2type;
}
operation Any initDomainInitValue() : Map {
        var domain2initValue : Map;
        domain2initValue.put("DiffLMH","L");
        domain2initValue.put("DiffLMHE","L");
        domain2initValue.put("MinCost",0);
        domain2initValue.put("MinSkill",0);
        domain2initValue.put("MinTimePar",0.0);
        domain2initValue.put("MinTimeSeq",0.0);
        domain2initValue.put("PowerCons",0.0);
        domain2initValue.put("ProbSucc",1000);
        domain2initValue.put("ReachPar",0);
        domain2initValue.put("ReachSeq",0);
        domain2initValue.put("SatOpp","true");
        domain2initValue.put("SatProp","true");
        domain2initValue.put("SatScenario","true");
        return domain2initValue;
}
// Returns an ATNode, based on its label
```

```
operation Any getATNode(target : String) : AT!Node {
        var nodes = AT!Node.allInstances.select(n|
           n.Children.size() == 0);
        for(node in nodes) {
                if(target.equals(node.Label)) return
                   node;
        }
        return null;
}
operation AT!Node getLeftSiblingsActivator() : String {
        // If there is no PAND gate
        if ( AT!Node.allInstances().select( n |
           n.Connector.isTypeOf( AT!PAND ) ).size() ==
           0 ) {
                return "";
        } else if ( not self.hasPandAncestor() ) {
                return "";
        } else {
                var result =
                   self.getLeftSiblingsActivatorHelper();
                if ( result.equals("LEFT") ) { return
                   ""; }
                return result;
        }
}
// Possible infite loop, if a node has itself as a
   parent or ancestor.
// The EVL constraints should prevent this.
operation AT!Node hasPandAncestor() : Boolean {
        if ( self.Parents.size() == 0 ) {
                return false;
        } else if ( self.Parents.select( n |
           n.Connector.isTypeOf( AT!PAND )).size() > 0
           ) {
                return true;
        } else {
                var result = false;
                for ( n in self.Parents ) {
                        if (n.hasPandAncestor()) {
                                result = true;
                        }
                }
                return result;
        }
}
// @return a sequence of all the first encountered
   PANDs
operation AT!Node getFirstPandAncestor() : Sequence {
        var result : Sequence;
        if ( self.Parents.size() == 0 ) {
                return result;
        } else if ( self.Parents.select( n |
           n.Connector.isTypeOf( AT!PAND )).size() > 0
           ) {
                for ( parent in self.Parents ) {
                        if (parent.Connector.isTypeOf(
                           AT!PAND ) ) {
                                result.add(parent);
                        }
                }
                return result;
        } else {
                var result = false;
                for ( n in self.Parents ) {
                        if (n.hasPandAncestor()) {
                                result = true;
                        }
                }
                return result;
        }
}
operation AT!Node getLeftSiblingsActivatorHelper() :
   String {
        if (self.Parents.size() == 0) {
                return "";
        } else {
                for ( parent in self.Parents ) {
```

```
                                if (
                                   parent.Connector.isTypeOf(
                                   AT!PAND ) ) {
                                        var children =
                                           parent.Children;
                                        var index =
                                           children.indexOf(self)
                        .asInteger();
                                        if (index == 0) {
                                                return "LEFT";
                                        } else {
                                                return
                                                   children.at(index
                                                   -
                                                   1).getConvertedLabel();
                                        }
                                }
                        }
                        for ( parent in self.Parents ) {
                                var result =
                                   parent.getLeftSiblingsActivatorHelper();
                                if (not result.equals("")) {
                                   return result; }
                        }
                        // Finally since nothing was found
                        return "";
                }
        }
        %]
```

9.4 Appendix D: ATMM2Uppaal import

SystemTempalte.egl

```
        <template>
                <name>System</name>
                <parameter>broadcast chan &amp;a</parameter>
                <declaration>clock global_clk;</declaration>
                <location id="id0" x="-200" y="0">
                </location>
                <location id="id1" x="0" y="0">
                        <name x="-10" y="25">busy</name>
                </location>
                <location id="id2" x="200" y="0">
                        <name x="190" y="25">done</name>
                        <committed/>
                </location>
                <init ref="id0"/>
                <transition>
                        <source ref="id1"/>
                        <target ref="id2"/>
                        <label kind="synchronisation" x="17"
                           y="8">a?</label>
                        <label kind="assignment" x="17"
                           y="-42">system_done := true,
system_busy := false</label>
                </transition>
                <transition>
                        <source ref="id0"/>
                        <target ref="id1"/>
                        <label kind="assignment" x="-180"
                           y="-42">global_clk := 0,
system_busy := true</label>
                </transition>
        </template>
```

9.5 Appendix E: Templates for the Uppaal model

Exponential Basic Element

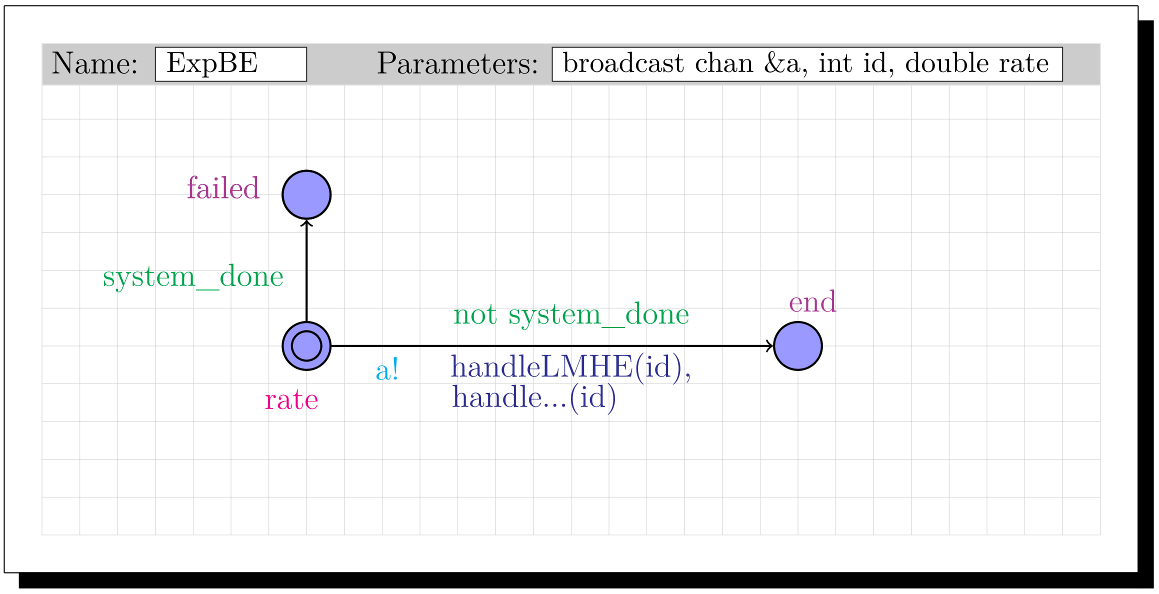


Figure 9.1: Basic Element (Trigger: Exp(1/e))

Erlang Basic Element

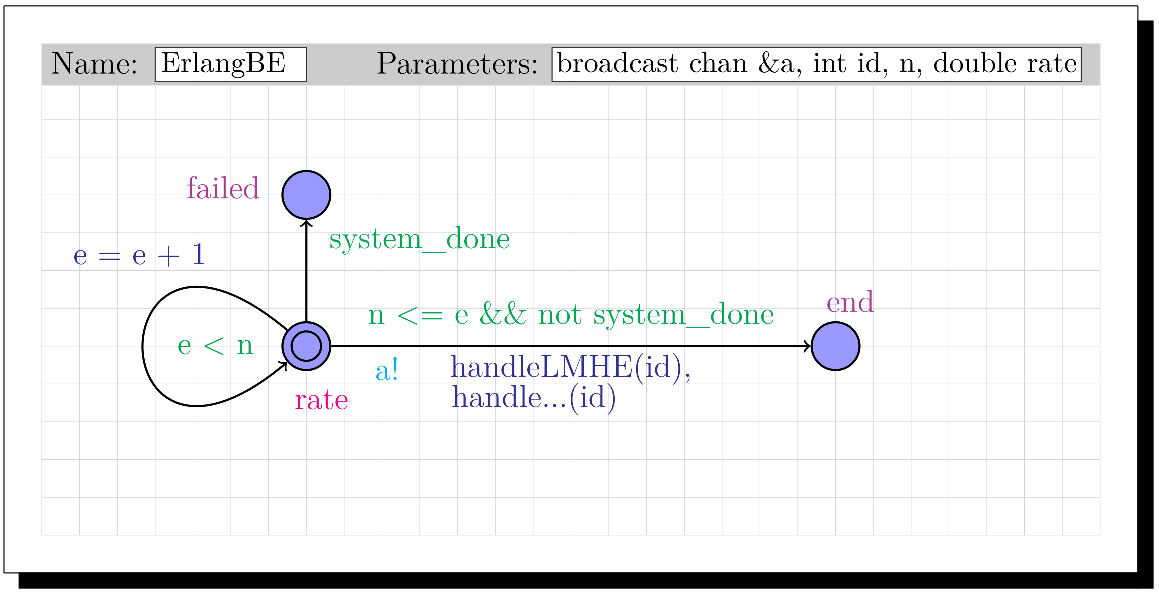


Figure 9.2: Basic Element (Trigger: Erlang(c, 1/d))

Timed Basic Element

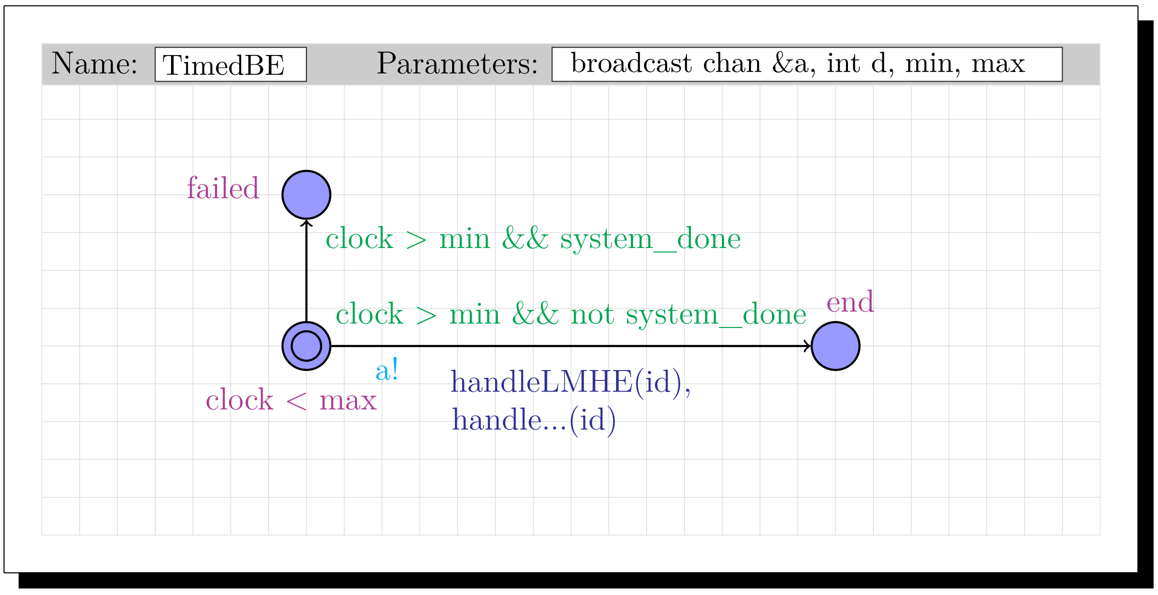
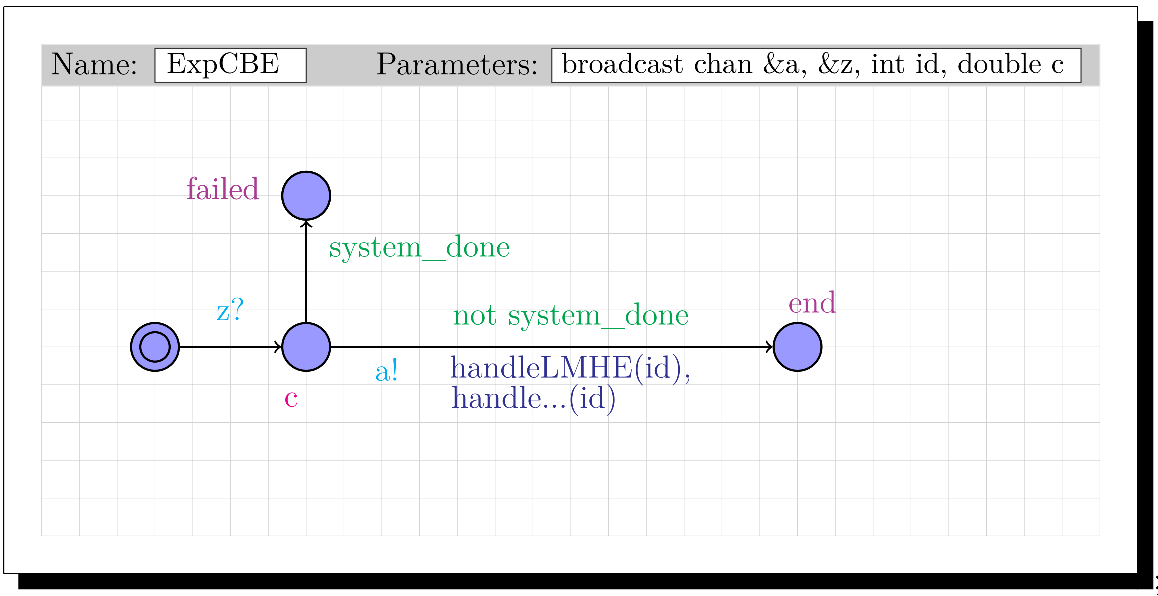

Name: TimedBE
Parameters: broadcast chan &a, int d, min, max
failed
clock > min && system_done
clock > min && not system_done
end
a!
handleLMHE(id),
handle...(id)
clock < max

Figure 9.3: Basic Element (Trigger: Time(c,d))

Dormant Exponential Basic Element



;

Figure 9.4: Dormant Basic Element (Trigger: Exp(1/e))

Dormant Erlang Basic Element

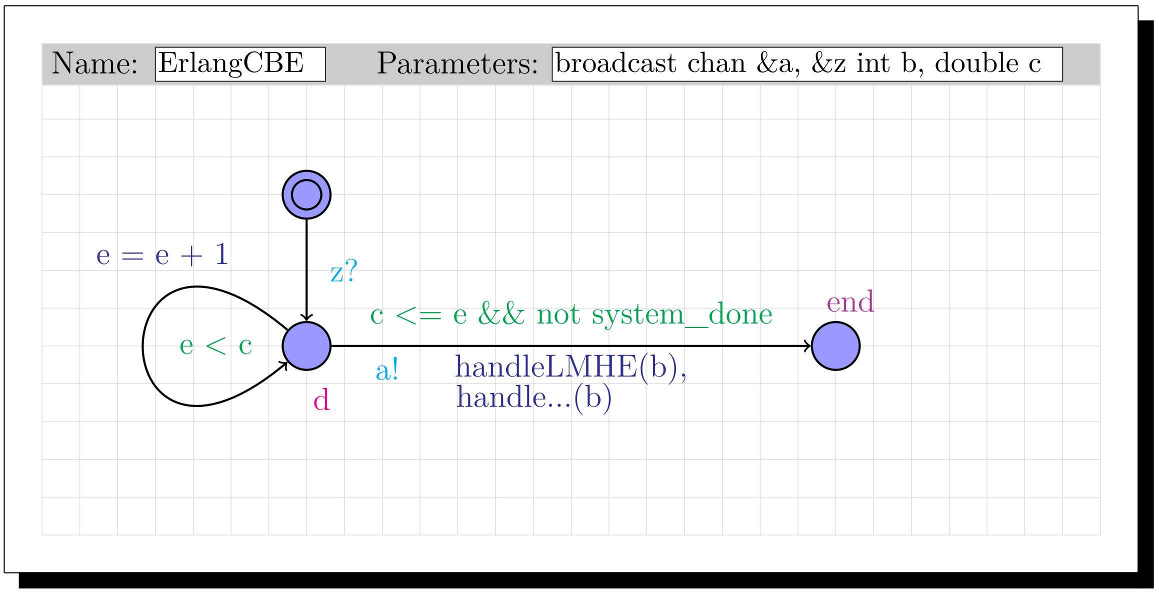

Name: ErlangCBE
Parameters: broadcast chan &a, &z int b, double c
e = e + 1
z?
e < c
c <= e && not system_done
end
a!
handleLMHE(b),
handle...(b)
d

Figure 9.5: Dormant Basic Element (Trigger: Erlang(c, 1/d))

Dormant Timed Basic Element

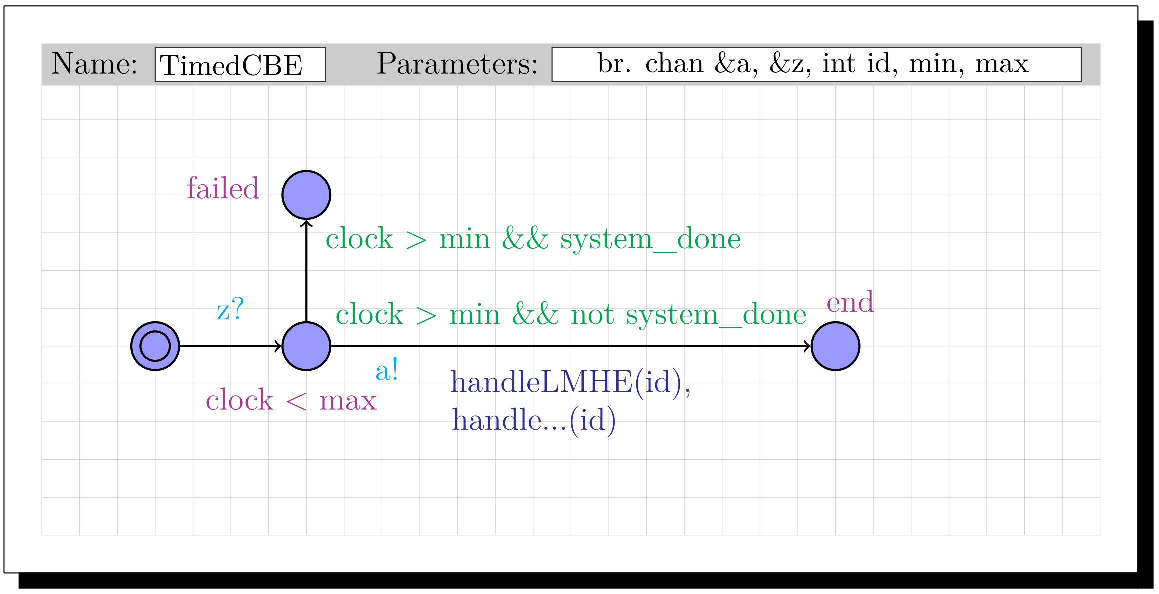

Name: TimedCBE
Parameters: br. chan &a, &z, int id, min, max
failed
clock > min && system_done
z?
clock > min && not system_done
end
a!
clock < max
handleLMHE(id),
handle...(id)

Figure 9.6: Dormant Basic Element (Trigger: Time(c,d))

AND-gate, with 2 children

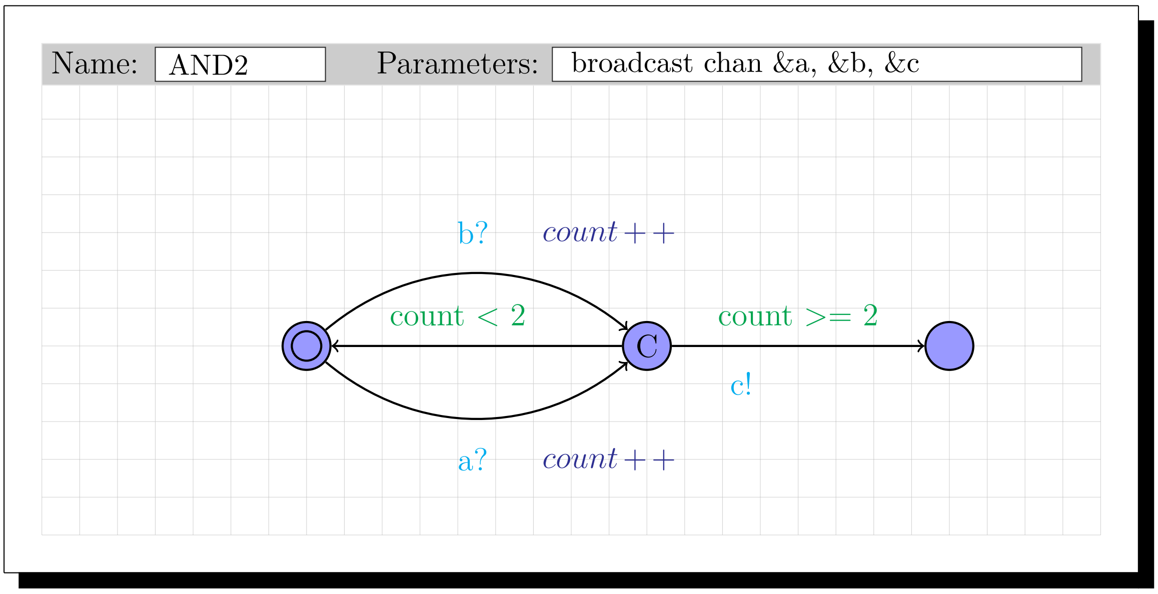


Figure 9.7: AND-gate, with 2 children

OR-gate, with 2 children

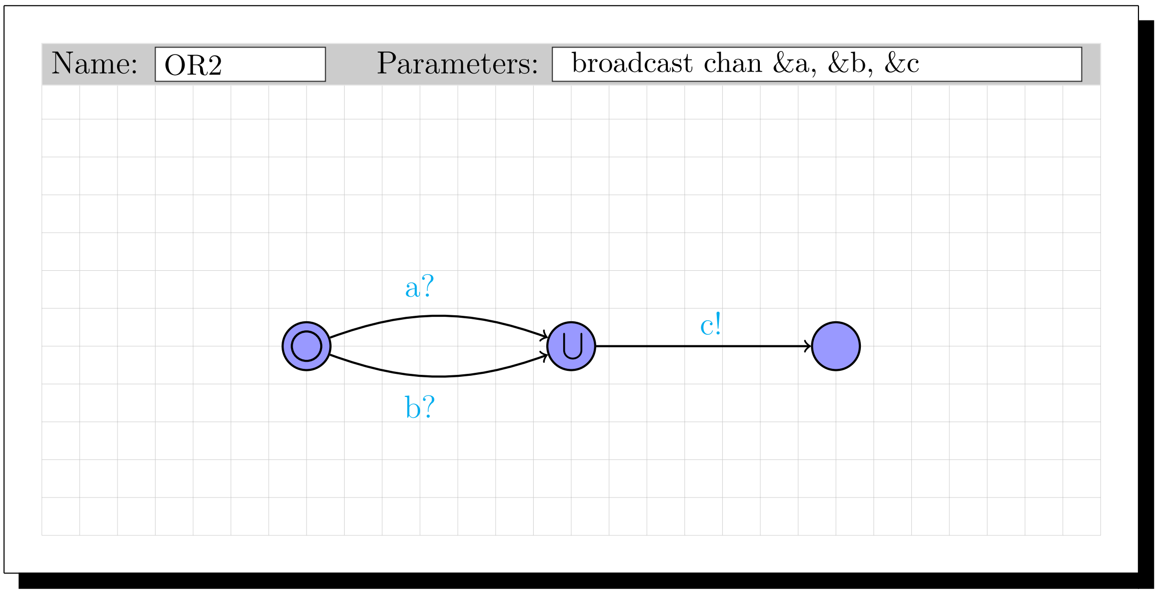


Figure 9.8: OR-gate, with 2 children

Dormant AND-gate, with 2 children

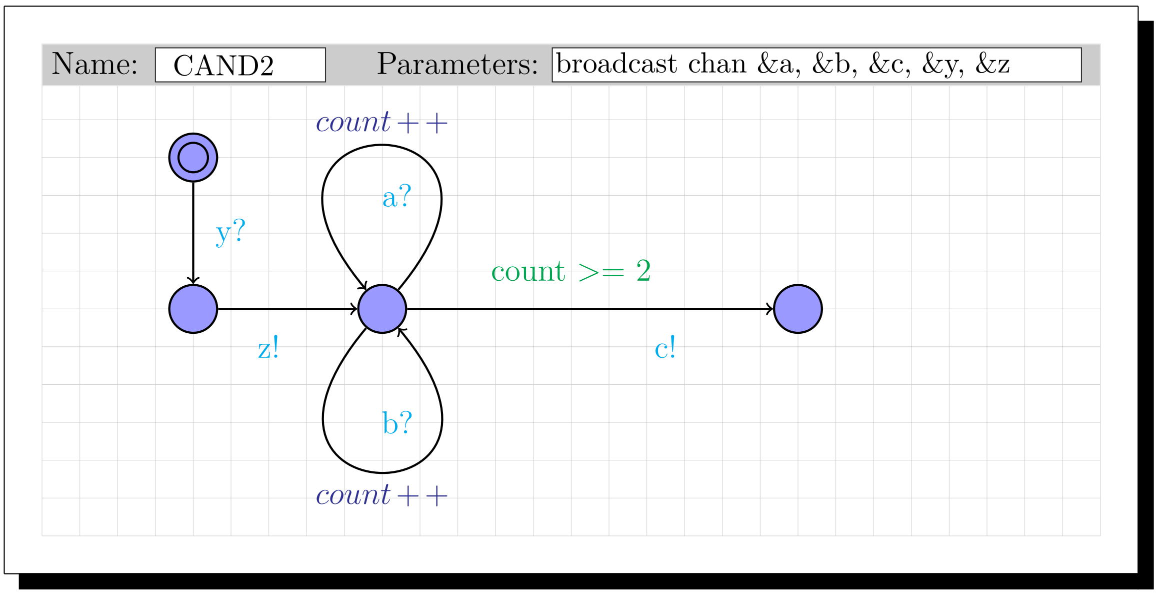


Figure 9.9: Dormant AND-gate, with 2 children

Dormant OR-gate, with 2 children

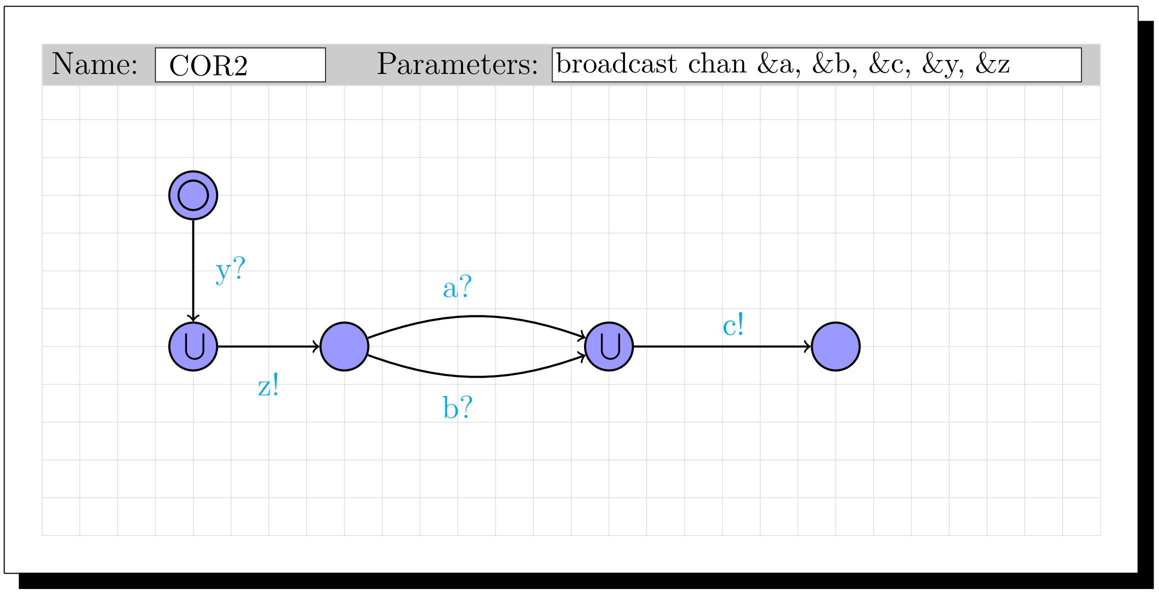


Figure 9.10: Dormant OR-gate, with 2 childre

9.6 Appendix F: Attack Tree Meta Model (ATMM)

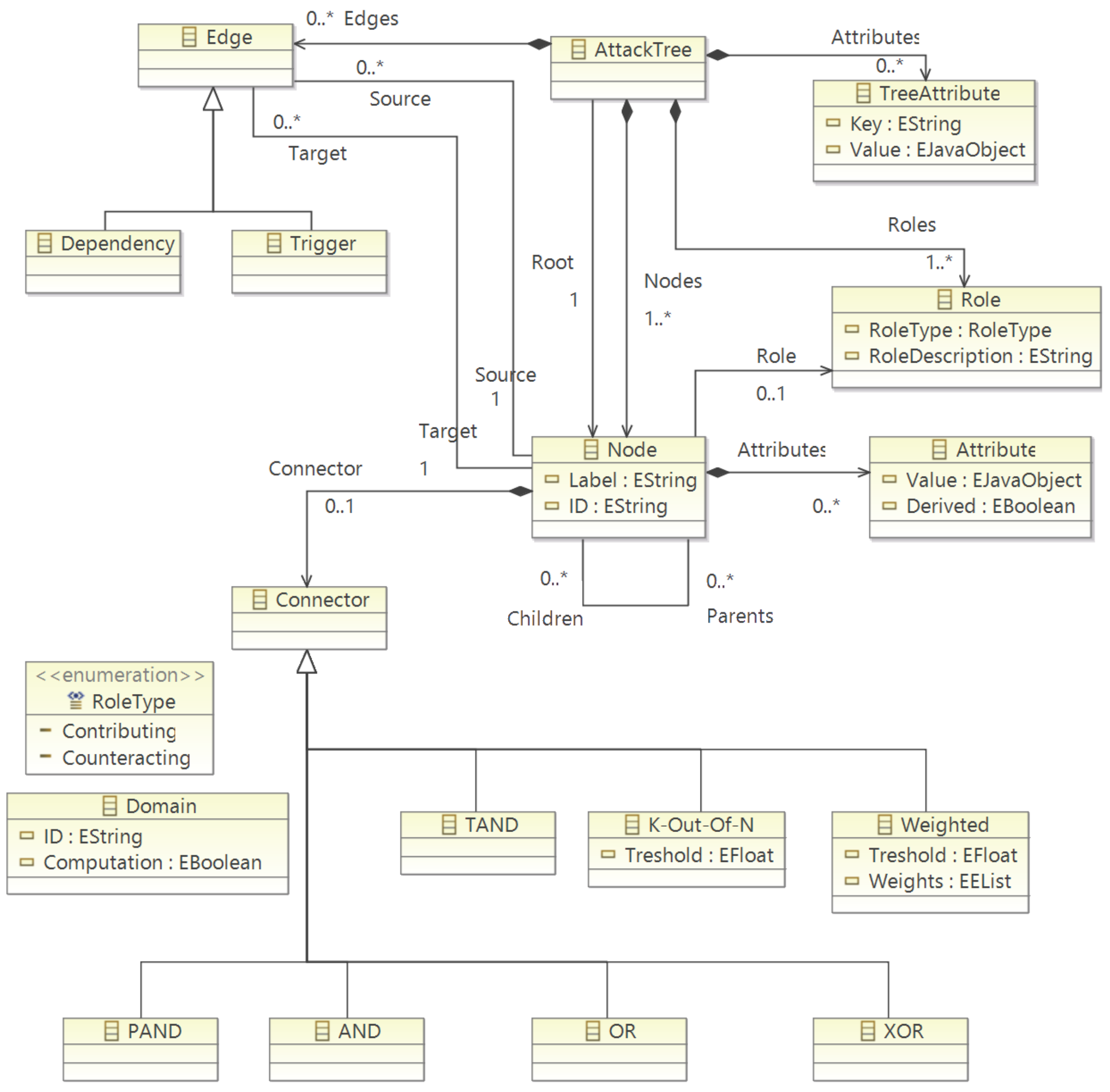


Figure 9.11: *Attack Tree Meta Model v1.0*

9.7 Appendix G: Java Standalone

```
package egl;
import *;

public class EglStandalone extends EpsilonStandaloneExample {

        private String output_url = "*/output/uppaal.xml";
        private boolean etl = false;
        private String input_url;

        public static void main(String[] args) throws Exception {
                if(args.length > 0) {
                        new EglStandalone(args[0]);
                } else {
                        System.out.println("Usage UATMM2Uppaal <input_file>");
                }
        }

        public EglStandalone(String input) {
                input_url = input;
                try {
                        System.out.println("");
                        setSource("transformation/ADTool2UATMM.etl");
                        this.execute();
                        etl = false;
                        System.out.println("Intermediate Model stored in:
                            model/Instance.model");
                        setSource("predefined/template.egl");
                        this.execute();
                } catch (Exception e) {
                        System.out.println("Some error occurred during the reading
                            of the files");
                        e.printStackTrace();
                }
        }

        @Override
        public IEolExecutableModule createModule() {
                if (etl) {
                        return new EtlModule();
                } else {
                        return new EglTemplateFactoryModuleAdapter(new
                            EglTemplateFactory());
                }
        }

        @Override
        public List<IModel> getModels() throws Exception {
                List<IModel> models = new ArrayList<IModel>();
                if (etl) {
                        models.add(createXmlModel("ADTool", input_url));
                        models.add(createEmfModel("UATMM", "model/Instance.model",
                            "model/UATMM.ecore", false, true));
                } else {
                        models.add(createEmfModel("AT",
                            "model/Complete_Tree_MM_Instance.model",
                            "model/UATMM.ecore", true, false)); //DK
                }
                return models;
        }

        @Override
        public String getSource()  {
                return resource;
        }

        @Override
        public void setSource(String s)  {
                resource = s;
        }

        @Override
        public void postProcess() {
                if (!etl) {
                        PrintWriter writer;
                        try {
                                writer = new PrintWriter(output_url, "UTF-8");
                                writer.println(result);
                                writer.close();
                        } catch (FileNotFoundException |
                            UnsupportedEncodingException e) {
                                System.out.println("The file required cannot be
                                    found or has an incorrect encoding");
                                e.printStackTrace();
                        }
                }
        }
}
```