Understanding Attacks: Modeling the outcome of Attack Tree analysis

J.H. Brandt
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.h.brandt@student.utwente.nl

ABSTRACT

Attack trees are used in structured approaches to describe all actions that an attacker must undertake to achieve a certain (malicious) goal. The analysis of such an attack tree provides insight in the vulnerabilities of this system. One way of attack tree analysis is the transformation of the attack tree into a priced timed automaton, and analysing this automaton with Uppaal. This approach generates a large amount of trace data, which is proof of the results that Uppaal provides. This trace data is undocumented and needs further parsing in order to gain usable information. These drawbacks make the isolation of information relevant to the attack tree rather cumbersome. This paper proposed a meta-model to model Uppaal and Uppaal trace data and shows that it is possible to build a compiler that compiles raw trace data into instances of this meta-model effectively. This paper also suggests how to derive insightful information and map this onto the original attack tree. This yields the original attack tree enriched with information derived from the trace data.

Keywords

Attack Tree, analysis, security, vulnerabilities, countermeasures, attack, Uppaal, Uppaal CORA, libutap, state trace, eclipse modeling framework, timed automata, TA, meta-model

1. INTRODUCTION

To understand the level of safety of computer systems or even physical systems, one needs to have knowledge about the possible threats. Attack trees are used in structured approaches to describe all actions that an attacker must undertake to achieve a certain (malicious) goal, such as hacking a website or breaking into a vault. The analysis of such an attack tree provides insight in the vulnerabilities of this system and the resources required for a successful attack on the system being modeled. The information acquired in analysis can be used to determine countermeasures for such attacks.

This paper focuses on one method for attack tree analysis, namely the transformation of the attack tree into a timed automaton (TA) and analysing this automaton with Uppaal, as showed by Kumar, Ruijters and Stoelinga in [8]. Uppaal is described as a tool that finds the cheapest path to a state satisfying certain goal conditions in an TA [2]. Besides the properties of the end state, this analysis also generates a large amount of trace data. This trace data contains the proof of the provided results. This trace contains information which can be used to increase the information represented in the attack tree. However, the raw trace data needs further parsing in order to gain usable information. These drawbacks make the isolation of information relevant to the attack tree rather cumbersome, and therefore this possible source of information currently remains unused.

Figure 2 shows the steps proposed to parse the trace data into a generic model and map this data onto the original attack tree. These models are instances of ecore meta-models, except for 'Native Uppaal trace data', which is encoded text.

However, due to unexpected issues with the first step there was not enough time left to finish the last step, details and further recommendations are discussed in section 9

Section 2 gives a more precise description of the problem and provides requirements for a solution, followed by sections 3 and 4 describing respectively background knowledge and related work. Sections 5 and 6 discuss the details of steps 3 and 4, whose validations are described by section 7. Section 8 gives a discussion, followed by section 9 containing the conclusion.

2. PROBLEM DEFINITION

The goal of this paper is to investigate the possibilities of tracing useful information from attack tree analysis back into a visually representable attack tree. More concretely, this project tries to answer the following question: How is the trace output of Uppaal usable to gain new information when analysing attack trees? An example of a (simple) attack tree can be found in figure 1.

Figure 2 gives an overview of all steps required in the scope of this project. Step 1 represents the conversion of an existing Attack tree model into a model that Uppaal can analyse, i.e. a native Uppaal model. Step 2 represents the attack tree analysis. Details of steps 1 and 2 are beyond the scope of this paper. It is enough to realize that this analysis method, like any other analysis using Uppaal, produces a state trace. This state trace describes which (TA-)transitions were traversed to find a valid end state, satisfying certain goal conditions. The states also contain the values for variables described by the original attack tree, and for other variables added for the purpose of the analysis. The state trace of Uppaal looks like the example shown in appendix A. Step 3 and 4 are described in the following two subsections.

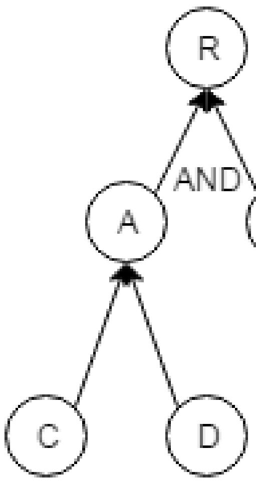

Figure 1. An example of an attack tree without attributes. Node R represents the goal of the attacker, nodes B, C and D represent Basic Attack Steps and node A represents a goal. Whenever the type of gate is omitted, it is an OR-gate.

2.1 Interpretation of the state trace

Step 3 describes the translation of this "Uppaal-trace" model into a usable variant that still covers the same data. To promote reusability, this project should also provide a solution to this step that will apply for all models that can be analysed using Uppaal. This brings an additional issue, as trace data of Uppaal is encoded, as shown in section 5.1 figure 5. This step also leads us to the first sub-question: How can we compile the output format of Uppaal into usable data?

If this trace data is decoded, the next steps should be similar to those applied on a trace of Uppaal. This means that the resulting data should be modeled in such a way that the provided solution is also usable on Uppaal models from other scientific domains. This leads us to the next sub-question: How can we model Uppaal trace data in a usable model?

2.2 The enrichment of the original attack tree

Step 4 describes the process of analysing the solution found in step 3 and extracting data that can be consider useful in the context of attack trees. If this data is available, it should be modeled into the original attack tree. The amount and the kind of information that can be learned depends on the information that is outputted in the state trace, thus depends on the information that is contained in the attack tree before analysis.

2.3 Requirements

The proposed solution should satisfy the following requirements.

- *Correctness.* It should be correct, thus not show any data that cannot be found in the trace, and should be complete;
- *Scalability and performance.* It should be scalable and not take longer than the duration of the analysis that generated the trace.
- *Extensibility.* it should be extensible to promote reuse of the provided solution in other contexts.
- *Fitness for purpose.* Just as parts of the solution should be usable in other contexts, the solution should fit its purpose. It should enrich an attack tree with new knowledge, based on the trace data gained during the analysis.

Section 7 discusses to what degree these conditions are checked .

3. BACKGROUND KNOWLEDGE

3.1 Attack Trees

Attack trees (ATs) are used to model the available actions of an attacker of a system. There is no single definition of attack trees: since a basic definition was proposed by Schneider [15], a lot of extensions have been studied. A broad overview of these extensions is given by Kordy et al. [7]. This project uses the definition of attack trees from Kumar, Ruijters and Stoelinga [8].

An AT is a directed acyclic graph [9] that is composed of basic attack steps (BAS) and goals. A BAS can be seen as an atomic step of an attack that an attacker can undertake, and they are the leaves of an AT. Goals represent an aggregation of BAS's and other goals and cannot be a leaf of an AT, so each should have at least one child. During analysis, each BAS or goal can have a state of either achieved or not achieved.

Each goal also contains one operator, the so-called gate, which specifies the relation between the states of it children and the state of the goal itself. An AT has 4 types of gates, as given by definition 1, AND-gates, OR-gates, SAND-gates and SOR-gates. AND-gates require all children to be achieved before the subgoal is achieved itself, where-as an OR-gate only requires one child to be achieved. Sequenced AND-gates (SAND) and Sequenced OR-gates (SOR) mostly resemble their unsequenced variants, they however differ in the chronological order that children should be achieved. Where this order is irrelevant for unsequenced gates, sequenced gates will only succeed if the children of the goal are achieved in the order specified by their sequence [5].

Given that the root element of an AT, or target of the attacker, has the same semantics as any other goal. We define the root element to be a goal that is not a child of any other goal.

An example of such attack tree can be found in section 2, figure 1.

Definition 1. The set of attack tree gate types is given by $GateTypes : \{AND, OR, SAND, SOR\}$

Furthermore, during attack tree analysis, an AT is executed by a specific attacker. Such attacker is described by an attacker profile, which is a fixed set of attributes. Attributes can specify, for example, a skill-level or available budget. We define an attacker profile *Profile* according to definition 2.

Definition 2. An attacker profile is a fixed set of real-valued attributes $Profile : \{a_1 ... a_n\} where\, a_i \in R$

Each BAS has a special attribute, *Time*, that represents time spent to achieve this BAS. Additionally, each BAS also contains two preconditions(*Enable, CanSucceed*) and a resulting effect(*Effect*). The preconditions are Boolean combinations of linear equations of attributes. *Enable* is a function over $\{a_1 ... a_n\}$, whereas *CanSucceed* is a function over $\{Time, a_1 ... a_n\}$. Both preconditions always evaluate

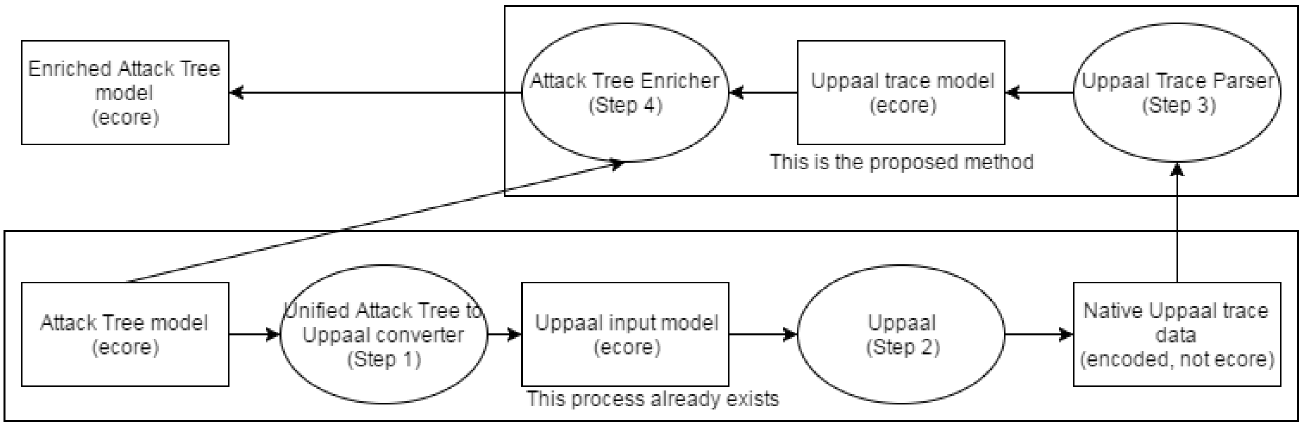
Enriched Attack Tree model (ecore)
Attack Tree Enricher (Step 4)
Uppaal trace model (ecore)
Uppaal Trace Parser (Step 3)
This is the proposed method
Attack Tree model (ecore)
Unified Attack Tree to Uppaal converter (Step 1)
Uppaal input model (ecore)
Uppaal (Step 2)
Native Uppaal trace data (encoded, not ecore)
This process already exists

Figure 2. An overview of the entire process. Rectangles represent data-instances, Ovals represent processes. Step 1 represents the conversion of an attack tree to a model that Uppaal can evaluate. Step 2 represents the analysis of an attack tree, the specifics of this step are out of scope for this paper. Step 3 represents the parsing and modeling of the Uppaal trace data into a generic Uppaal trace model. Step 4 represents the enrichment of the original attack tree with information gained from Step 3 (and is not finished because of time constraints).

to either $0(\mathit{false})$ or $1(\mathit{true})$, which indicates whether a specific BAS is enabled, respectively can be achieved. The resulting effect *Effect* updates the attacker profile *Profile* when the BAS is achieved. *Effect* is a function of all attributes $\{a_1 ... a_n\}$ over all values of these attributes, over the elapsed time *Time*, or $\mathit{Effect} : \{a_1 ... a_n\} \times \mathit{ValuesOf}\{a_1 ... a_n\} \times \mathit{Time} \rightarrow \mathit{newValue}\{a_1 ... a_n\}$. This specification leads to definition 4.

An example of attribute calculations can be found in example 1.

Example 1. Example of attribute calculations

Given attacker profile $\mathit{Profile} : \{\mathit{budget}, \mathit{skill}\}$ valued at $\{1500, 50\}$, and given a BAS with following functions:
$\mathit{Enabled}(\{\mathit{budget}, \mathit{skill}\}) : \mathit{skill} \geq 50$
$\mathit{CanSucceed}(\{\mathit{Time}, \mathit{budget}, \mathit{skill}\}) : (\mathit{Time} > 15) \wedge (\mathit{budget} > 200)$
$\mathit{Effect}(\mathit{budget}, \{\mathit{Val}_{budget}, \mathit{Val}_{skill}\})(\mathit{Time}) : \mathit{val}_{budget} - \mathit{Time} * (100 - \mathit{skill})$
$\mathit{Effect}(\mathit{skill}, \{\mathit{Val}_{budget}, \mathit{Val}_{skill}\})(\mathit{Time}) : \mathit{val}_{skill}$

The initial ($\mathit{Time} = 0$) preconditions evaluate as follows:
$\mathit{Enabled}(\{1500, 50\}) = 50 \geq 50 = 1$
$\mathit{CanSucceed}(\{0, 1500, 50\}) = 0 > 15 \wedge 1500 > 200 = 0$

After some time elapsed, $\mathit{Time} = 20$.
$\mathit{CanSucceed}(\{20, 1500, 50\}) = 20 > 15 \wedge 1500 > 200 = 1$
$\mathit{Effect}(\mathit{budget}, \{1500, 50\})(\mathit{20}) = 1500 - 20 * (50) = 500$
$\mathit{Effect}(\mathit{skill}, \{1500, 50\})(\mathit{20}) = 50$

From this follows the new attacker profile, valued at $\{500, 50\}$.

Definition 3. An basic attack step (BAS) is a tuple (Time, achieved, Enable, CanSucceed, Effect), where

- Time *represents the time elapsed since the start of the BAS*
- achieved : $\{1, 0\}$ *indicates whether or not the BAS is achieved*
- Enable : $\{a_1 ... a_n\} \times \{1, 0\}$ *indicates whether or not the BAS is enabled*
- CanSucceed : $\{\mathit{Time}, a_1 ... a_n\} \times \{1, 0\}$ *indicates whether or not the BAS can be achieved*
- Effect : $\{a_1 ... a_n\} \times \mathit{ValuesOf}\{a_1 ... a_n\} \times \mathit{Time} \rightarrow \mathit{newValue}\{a_1 ... a_n\}$ *is the update function that gets invoked when the state is achieved*

Attributes $\{a_1 ... a_n\}$ refer to attributes of the attacker profile, explained in definition 2.

Definition 4. An attack tree AT is a tuple (BAS, Goal, Target, Relation, Gate), where

- BAS *is a finite set of basic attack steps, as defined by definition 3*
- Goal *is a finite set of goals, disjoint from BAS*
- Target $\in$ Goal *is the attacker goal*
- Rel : $\mathit{Goal} \rightarrow (\mathit{Goal} \setminus \{\mathit{Target}\} \cup \mathit{BAS})^*$ *is the mapping of a goal to its children*
- Gate : $\mathit{Goal} \rightarrow \mathit{GateTypes}$ *is the assignment of gates to goals, as specified in definition 1*

3.2 Timed Automata

Finite state automata (FSA) are the most basic type of automata used to model and analyze systems. However an FSA does not provide functionality to describe real-time systems. This is where timed automata (TA) come into play: they extend the basic FSA by adding a finite set of clocks that increase during evaluation and which can be used as transition constraints. [1]

A (simplified) example of a timed automaton can be found in figure 3. The fastest trace can be found in example 2.

Uppaal is a verification tool for TA, and is used in both academia and industry as model checker [2].

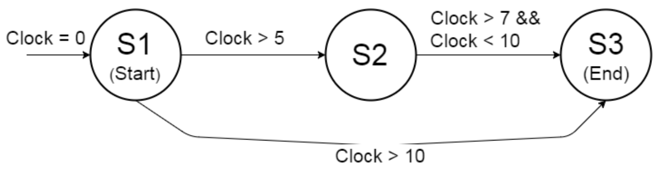

Figure 3. An example of a timed automaton. This example does not have any real application as it is simplified to be an understandable example.

Example 2. Example of an TA trace for figure 3

```
State S1 Clock = 0

Delay: 6
State S1 Clock = 6

Transition: S1->S2 {Clock > 5}
State S2 Clock = 6

Delay: 2
State S2 Clock = 8

Transition S2->S3 {Clock > 7 && Clock < 10}
State S3 Clock = 8
```

3.3 Model-driven development

Model-driven development is the development of applications using meta models. A meta-model is a model which describes a group of models. For example, a meta-model describing buildings can state that any building consists of multiple rooms and that those rooms contain doors and windows. An instance of this meta-model is a specific building, and can be referred to as a model of a building or as an instance of a meta-model of a building.

3.3.1 Eclipse Modeling Framework

Eclipse modeling framework (EMF) is a framework that provides modeling and code generation tools [16], only the features relevant for this project will be explained in this section. The framework uses .ecore-files to define meta-models and generate Java-code that corresponds to this definition. EMF also provides an API to programatically create new instances of these meta-models in Java or ETL(see next subsection). It is possible to define a meta-model so that it re-uses certain properties defined in another meta-model, this will cause the instance of the meta-model to have references to instances of an other meta-model.

3.3.2 Epsilon

Epsilon is a toolset containing different kind of tools and languages that can be used for modeling-related activities, including, but not limited to, modeling itself, model-to-model transformation and code-generation [11]. One of these languages is the Epsilon Transformation Language, described below.

Epsilon Transformation Language

Epsilon Transformation Language (ETL) is a rule-based model-to-model transformation language and is used to specify the rules that are used to define the relation between an instance of one meta-model into an instance of another meta-model [6]. These rules can be evaluated by Epsilon to transform the first model into the latter, this makes use of the API provided by EMF to create a new instances of a meta-model.

4. RELATED WORK

Sampaio describes a similar application, he describes the user-friendly visual interface to manage industry control systems for process automation [13]. Sampaio traces information during Uppaal analysis of TA, by naming conventions in both original models and Uppaal intermediate models. He analyses the result trace in case of a fail, in order to identify which software module is responsible for the faulty condition. Havelund et al. modeled a Bang & Olufsen audio/video protocol in Uppaal to find an error in this protocol [4]. They detected that their model reached a state unreachable with a valid protocol, which indicated that there was indeed an error. After finding the shortest trace to reach this state, Uppaal produced a state trace containing 1998 transition steps. The mistake was found by searching those transitions using an intelligent development environment. The authors however state that this approach was tedious and error-prone and argue that another approach is desirable.

5. STEP 3: UPPAAL TRACE PARSER

Step 3 of figure 2 is the compilation of Uppaal trace output into a format which makes sense, this task is split up in three parts. Figure 4 shows a more detailed overview of step 2 and step 3.

5.1 Uppaal encoding

Part 1 of figure 4 is the decoding of the Native Uppaal trace data. This unprocessed trace data, as shown in figure 5, has all string replaced by references to a specific model that is used internally by Uppaal, the Uppaal internal format. As there is no documentation available on this model, this step should be addressed using existing solutions. To achieve this, this paper uses the library named libutap, Uppaal Timed Automata Parser LIBrary [10]. Libutap transforms the Native Uppaal trace data into Human readable Uppaal trace data, using the Uppaal internal format. As libutap that contained numerous errors, these were resolved with help of the author. An example of the output of the (fixed) libutap can be found in appendix A.

5.2 Trace meta-model

The concrete data from the previous part should be used in a logic model, this leads to part 2: the design of an Uppaal trace model. There is a generic meta-model available, created by Gerking [3], that can be used to create a model of Uppaal input. This paper proposes a meta-model describing Uppaal trace data in a way that it links properties that are already defined in the existing meta-model to this new meta-model. This means that instances of the Uppaal trace meta-model have references to (properties of) instances of the Uppaal input meta-model. This new Uppaal trace meta-model will be defined as an .ecore-file in EMF.

However, during development of the parser (described in section 5.3) it turned out that the Java interface provided by EMF is not able to load instances of the (already existing) Uppaal meta-model. Extensive debugging and testing showed that the this interface failed to recognize the type of class of the entities in the model. The so-called *ClassifierIDs*, which is used to find the class of a model entity, mismatched the IDs in the model classes generated

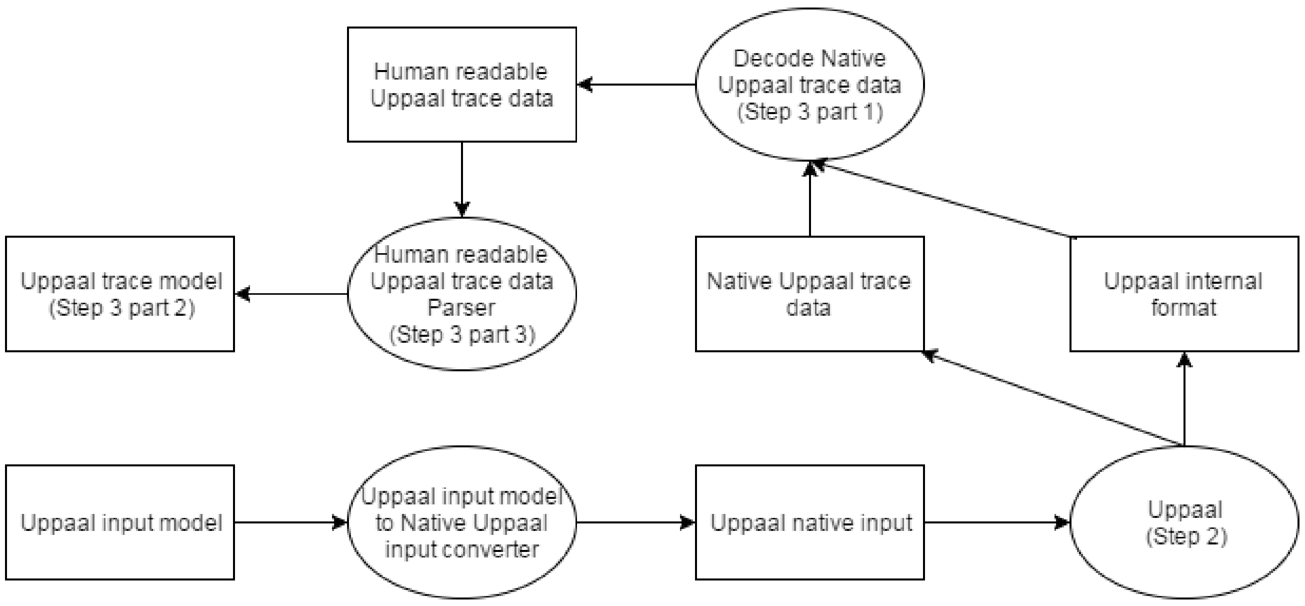
Human readable Uppaal trace data
Decode Native Uppaal trace data (Step 3 part 1)
Uppaal trace model (Step 3 part 2)
Human readable Uppaal trace data Parser (Step 3 part 3)
Native Uppaal trace data
Uppaal internal format
Uppaal input model
Uppaal input model to Native Uppaal input converter
Uppaal native input
Uppaal (Step 2)

Figure 4. A detailed overview of steps 2 and 3. Rectangles represent data-instances, Ovals represent processes. Data-instances ending in 'model' represent instances of .ecore meta-models. Step 3 has been split up in parts, ordered by the order that these parts are mentioned in section 5.

```
2 0 0 2 . 0 1 0 . 1 2 0 . 2 3 0 . 3 0 0 . . 0 1 0
5 6 0 10 12 1 0 0 0 0 0 . 1 0 0 2 . 0 1 0 . 1 2 0
. 2 3 0 . 3 1 0 . . 0 1 0 5 6 0 10 12 1 0 0 0 0 0
. 0 2 . 1 0 3 2 . 0 3 0 . 1 2 0 . 2 1 0 . 3 0 25
. 3 1 0 . . 0 1 0 5 6 0 10 12 1 0 0 0 0 0 . 2 1 .
1 3 3 2 . 0 2 0 . 2 0 13 . 2 3 0 . 3 0 25 . 3 1 0
. . 0 1 0 5 6 0 10 12 1 0 0 0 0 0 . 1 1 . .
```

Figure 5. The format that trace-data of Uppaal adheres to. Newlines have been replaced with spaces to enhance readability.

by EMF. In order to work around this issue, we defined a new intermediate trace (ecore) meta-model that is equal to the definitive trace meta-model in every way except for the references to the Uppaal meta-model. All references to the Uppaal input meta-model are replaced by string attributes. The value of those attributes should be unique, e.g. a template name, so that future work can identify what these values refer to. The parser creates an instance of this intermediate meta-model.

A class diagram of the trace meta-model can be found in figure 6 and is built from the following components.

- Each trace is represented by an instance of class Trace.
- Each instance of Trace contains multiple instances of State, AbstractTransition and TemplateInstance
- Each TemplateInstance represents an instance of an Uppaal template and contains multiple instances of LocationInstance
- Each LocationInstance represent an instance of Uppaal locations.
- Each instance of an AbstractTransition can either be a DelayTransition or an EdgeTransition, representing respectively a time delay or (a list of) Uppaal edge(s) that have been traversed. Instances of AbstractTransition also have a source and a target, representing the instances of State that the AbstractTransition bridges.
- Each instance of a state represent snapshots of Uppaal model evaluation, and contains Valuations.
- Each instance of a Valuations represents the value of a certain variable at a specific moment, and contains an instance of Value. Furthermore each Valuation has references to instances of LocationInstance, TemplateInstance and also refers to the incoming and outgoing AbstractTransition.

The meta-model was designed this way in order to resemble the Uppaal input meta-model as much as possible. For this reason, there are a lot of different types of values. To keep the overview uncluttered, classes implementing the abstract class 'Value' have not been included in the description and in the class diagram.

5.3 Parser

The third part is the design of a parser that reads the Human readable Uppaal trace data, and parses it into a valid model Uppaal trace model (as defined by section 5.2). This parser is created using the parser-generator ANTLR [12] to read the trace data, and the appropriate API from EMF to generate the correct instance of the meta-model.

6. STEP 4: ATTACK TREE ENRICHER

Step 4 of figure 2 represents the extraction of useful information from the model generated by step 3, and the mapping of this information onto the original attack tree. As the attack tree meta-model is also defined as an .ecore-file, the model-to-model transformation language ETL can be used to transform data from the trace-data model to the attack tree model. The amount and the kind of information that can be learned depends on the information that is output in the state trace, thus depends on the information that is contained in the attack tree before analysis. The solution should be validated using two attack trees with different information.

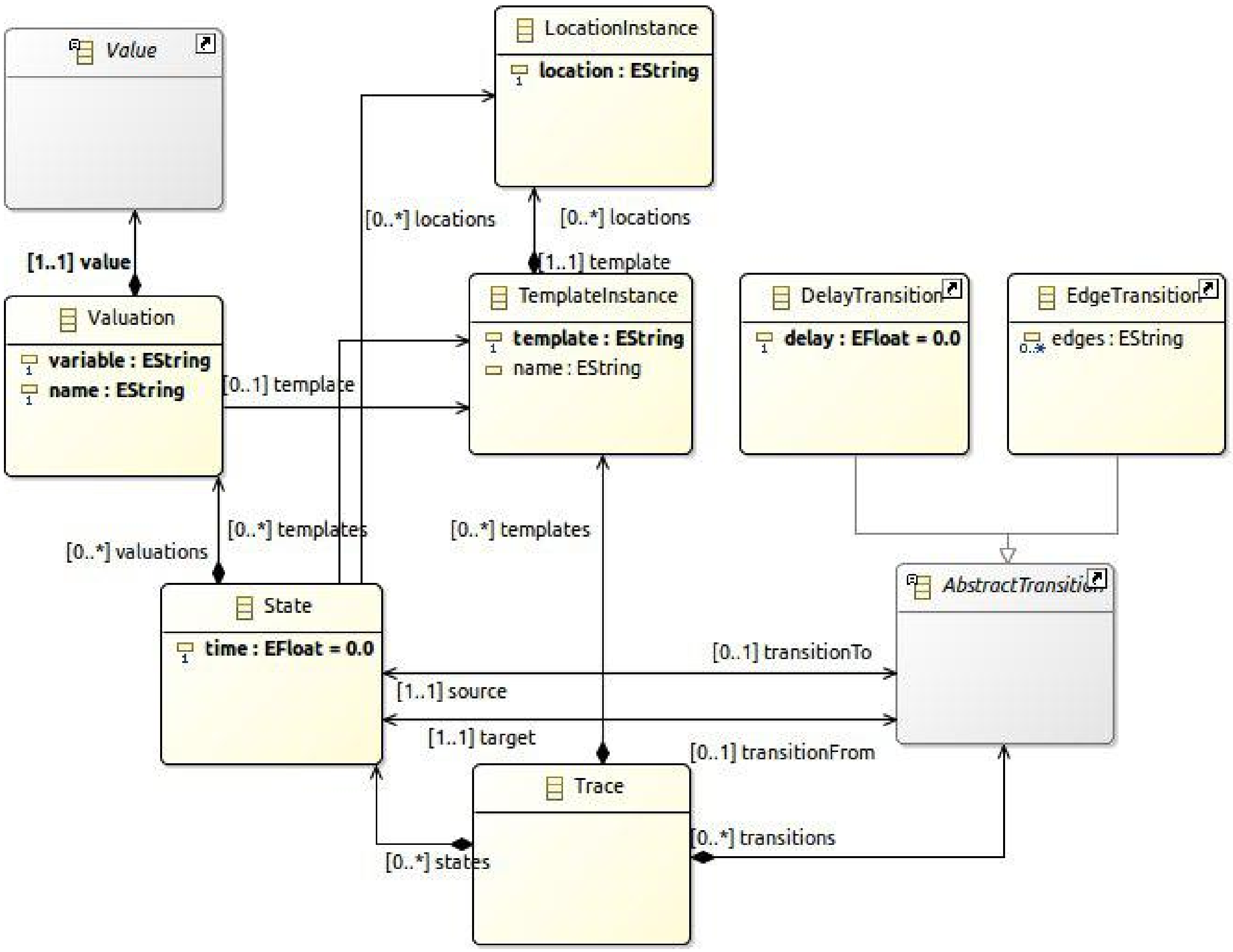

Figure 6. The class diagram of the designed trace meta-model. Implementations of abstract class Value are omitted to keep this overview accessible.

- An attack tree containing numerous attributes at each BAS, the solution should deduce the appropriate values at all goals the trace passes.
- An attack tree containing mostly sequenced gates. The solution should recognize that such attack tree does have a specified order of steps and apply this information to the model.

While attempting to create an ETL-script that does the transformations described in the list above, we ran into several issues.

- Eclipse Epsilon crashed every time moderately large trace files(about 50MB) were transformed.
- ETL was unable to load dependent meta-models. As the meta-model of the attack tree is defined as two separate meta-models(structure and values), one dependent on the other, this is an issue.

Because of these reasons and taking into account that step 3 took more time than expected, there was not enough time left to finalize step 4. However, given enough time the issues mentioned above should be solvable.

7. VALIDATION & EVALUATION

The requirements stated in section 2 are addressed in this section to validate whether or not the provided solution satisfies them. However, as not all problems are solved, some requirements could not be tested.

Section 7.3 specifies a requirement that needs to be tested in multiple scientific domains, this is to ensure that the solution is usable in the other applications. These different domains, are provided by researchers or PhD-students working with models in diverse domains at the University of Twente, Department of Formal Method and Tools. These 'different domains' are, besides the domain of Attack Trees, the domain of biological pathway modeling [14].

Along the set of models that are used for testing these requirements, there should ideally be at least one small and one large model from every different domain. The small model will be used to manually verify the results, whereas the big model can be used to verify performance. However, for some domains it was not feasible to acquire small models. In these cases a few trace states were taken from the larger model, parsed and their results verified manually.

7.1 Correctness

The provided solution will be tested to ensure that it is correct, this means that the solution should not put any information in the trace model that cannot be derived from the result trace. Furthermore, the solution should be complete, this means that it should transform all useful data from the trace back into the trace model.

This is tested manually using several different models, all tested models are shown in table 1, none failed.

7.2 Scalability

The provided solution should be scalable and not take longer than the duration of the (Uppaal) analysis that generated the trace. This is tested by acquiring different large Uppaal models, and benchmarking the time it takes to generate the generic (or model-specific) "Uppaal model".

Table 1. All models tested for correctness, results were verified manually. States means 'number of states'. All biological models have been stripped to 25 states, as described in section 7. Models with (CORA) annotation in their name are analysed using Uppaal CORA, this is explained later in this paper.

Name	States
Attack Tree #1	5
Attack tree #2	14
Attack tree (CORA) #1	5
Attack tree (CORA) #2	14
Biological model #1	25
Biological model #2	25
Biological model #3	25

Time measurements are taken by taking the times just before and right after the process measured is executed, using the method
`Java.lang.System.currentTimeMillis()`. Results are shown in table 2.

7.3 Extensibility

The solution provided should also have the ability to model trace output other scientific domains. The results of correctness and scalability testing, shown in tables 1 and 2, are sufficient to demonstrate this is the case.

7.4 Fitness for purpose

As explained in sections 5.2 and 6, due to time constraints 'step 4: The attack tree enricher' was not finished. Therefore this requirement cannot be verified.

8. DISCUSSION

Despite the fact that the entire solution is not fit for purpose, the result of step 3, the compilation of Uppaal trace data into a usable model, is usable in other research. As section 9 states, it seems fully compatible for models analysed by either Uppaal from different scientific domains. The biggest limitation seems to be the use of memory, which is the result of the building of the parse tree by ANTLR, as can be seen in table 2. ANTLR is configured to parse all information contained in the trace, while it might be less memory- and CPU-intensive to not parse unused and complex rules, such as the description of edges in the trace. As the description of the edge traversed is only saved entirely as unique identifier, parsing its contents seems like a waste of resources.

9. CONCLUSION

Correctness testing in section 7 shows that the solution works with models from different domain.

Scalability testing indicates that the solution requires a lot of memory: a trace file of 534MB cannot be parsed with 30GiB of memory available. Furthermore, it satisfies the requirement to compile faster than the analysis takes for attack tree models, but not for biological models. This is because Uppaal analysis of biological models is very fast, where-as Uppaal analysis of attack trees takes much longer as shown in table 2. As the duration of trace seems to scale linearly with the size of the trace file, the solution is scalable given that enough RAM is available.

Extensibility testing indicates that the provided solution can be used in other scientific domains, given that the trace data does not get too large or that enough memory is available.

Table 2. Scalability testing. Parse time represents the duration of our solution. Analysis time represents the duration of analysing the Uppaal model and generating the trace. Measurements have been taken using a machine with 16-core 2.8GHz Intel Xeon E5-2680 v2 and 30GiB of RAM, the JVM was configured to use all system resources if necessary. Uppaal analysis have been cut off after 15 minutes, as this clearly indicates that the requirement (Parse time is lower than Analysis time) has been met. Time intervals have been measured without the time it took to write the results to the file system. Total Parse time (duration of step 3) equals to Time ANTLR parsing + Time generating output.

Name	States	Trace file size (MB)	Analysis (s)	ANTLR parsing (s)	Output generation (s)
Biological model #1	60021	1100	6	Out of memory	-
Biological model #2	29790	534	6	Out of memory	-
Biological model #3	12875	230	6	604	116
Attack tree #1	757	26	> 900	50	10
Attack tree #2	290	10	> 900	18	3

The solution is not fit for its purpose, as within the scope of this paper there was no time to write step 4, the enrichment of the original attack tree.

The first research sub-question: ”How can we compile the output format of Uppaal into usable data” can now be answered. Section 5.1 shows that we can decode the Uppaal trace using the (open source) library libutap, providing us with a human-readable trace file. Furthermore, the most recent versions of Uppaal (but not Uppaal CORA) can provide trace data similar to the trace file generated by libutap. The provided solution has been extended to parse this trace data without issues.

Our second sub-research question: ”How can we model Uppaal trace data in a usable model?” can be answered using section 5.2. We cannot model the trace data based on the Uppaal input meta-model in Java, because EMF cannot not load instances of the Uppaal input meta-model. By removing the references to the Uppaal input model from the Uppaal trace model, it is possible to effectively model the trace data of both Uppaal. If one wants to use the original trace meta-model based on the Uppaal input meta-model, it should be possible to apply ETL to both the intermediate trace model and the Uppaal input model in order to generate an instance of the original trace meta-model. However, because of time constraints this could not be done in the course of this paper.

With the answer to these sub-questions, we address the main research question: ”How is the trace output of Uppaal usable to gain new information when analysing attack trees?”. As the sub-questions indicate, the trace data can be modeled in a structured manner. Furthermore the data contained in the trace seem to contain enough information to enrich attack trees using the patterns described in section 6. However, as there was not enough time to implement step 4, it cannot be concluded for sure that the described approach is effective to gain new information when analysing attack trees.

10. REFERENCES

[1] Alur, R., and Dill, D. L. A theory of timed automata. *Theor. Comput. Sci. 126*, 2 (Apr. 1994), 183–235.

[2] Behrmann, G., David, A., Larsen, K. G., Hakansson, J., Petterson, P., Yi, W., and Hendriks, M. *UPPAAL 4.0.* QEST ’06. IEEE Computer Society, Washington, DC, USA, Sept 2006, pp. 125–126.

[3] Gerking, C. Transparent uppaal-based verification of mechatronicuml models. Master thesis, University of Paderborn, May 2013.

[4] Havelund, K., Skou, A., Larsen, K. G., and Lund, K. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE* (Dec 1997), pp. 2–13.

[5] Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., and Trujillo-Rasua, R. Ict systems security and privacy protection: 30th ifip tc 11 international conference, sec 2015, hamburg, germany, may 26-28, 2015, proceedings. H. Federrath and D. Gollmann, Eds., Springer International Publishing, pp. 339–353.

[6] Kolovos, D. S., Paige, R. F., and Polack, F. A. C. *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. The Epsilon Transformation Language, pp. 46–60.

[7] Kordy, B. K., Piètre-Cambacédès, L., and Schweitzer, P. Dag-based attack and defense modeling: Don’t miss the forest for the attack trees. *Computer Science Review 13-14* (2014), 1–38.

[8] Kumar, R., Ruijters, E., and Stoelinga, M. *Formal Modeling and Analysis of Timed Systems: 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings.* Springer International Publishing, Cham, 2015, ch. Quantitative Attack Tree Analysis via Priced Timed Automata, pp. 156–171.

[9] Mauw, S., and Oostdijk, M. *Information Security and Cryptology - ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, ch. Foundations of Attack Trees, pp. 186–198.

[10] Mikucionis, M. Uppaal timed automata parser library, http://people.cs.aau.dk/ marius/utap/, 2015.

[11] Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., and Polack, F. A. C. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex*

Computer Systems (Washington, DC, USA, 2009), ICECCS '09, IEEE Computer Society, pp. 162–171.

[12] Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[13] Sampaio, L. R., Perkusich, A., and da Silva, L. D. Ladder programs validation through model-code traceability. In *Industrial Technology (ICIT), 2011 IEEE International Conference on* (March 2011), pp. 276–280.

[14] Schivo, S., Scholma, J., Wanders, B., Urquidi Camacho, R. A., van der Vet, P. E., Karperien, H. B. J., Langerak, R., van de Pol, J. C., and Post, J. N. Modelling biological pathway dynamics with timed automata. *IEEE Journal of Biomedical and Health Informatics 18*, 3 (May 2014), 832–839.

[15] Schneier, B. Attack trees. Dr. Dobb's Journal, Dec 1999.

[16] Steinberg, D., Budindky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2008.

APPENDIX

A. HUMAN-READABLE TRACE DATA, GENERATED BY LIBUTAP

```
State: Sys._id9 d_Door._id3 d_Window._id3
d_EnterRoom._id2 f = 0 t = 1 system_done = 0
d_Door.min = 5 d_Door.max = 6 d_Door.id = 0
d_Window.min = 10 d_Window.max = 12 d_Window.id =
1 t(0)-Sys.clk<=0 t(0)-d_Door.clk<=0
t(0)-d_Window.clk<=0 Sys.clk-d_Door.clk<=0
d_Door.clk-d_Window.clk<=0 d_Window.clk-t(0)<=0

Transition: Sys._id9 -> Sys.busy {1; 0; clk = 0;}

State: Sys.busy d_Door._id3 d_Window._id3
d_EnterRoom._id2 f = 0 t = 1 system_done = 0
d_Door.min = 5 d_Door.max = 6 d_Door.id = 0
d_Window.min = 10 d_Window.max = 12 d_Window.id =
1 t(0)-Sys.clk<=0 t(0)-d_Door.clk<=0
t(0)-d_Window.clk<=0 Sys.clk-d_Door.clk<=0
d_Door.clk-d_Window.clk<=0 d_Window.clk-Sys.clk<=0

Transition: d_Door._id3 -> d_Door._id6 {1; 0;
clk = 0;}

State: Sys.busy d_Door._id6 d_Window._id3
d_EnterRoom._id2 f = 0 t = 1 system_done = 0
d_Door.min = 5 d_Door.max = 6 d_Door.id = 0
d_Window.min = 10 d_Window.max = 12 d_Window.id =
1 t(0)-Sys.clk<=0 t(0)-d_Door.clk<=0
t(0)-d_Window.clk<=0 Sys.clk-d_Window.clk<=0
d_Door.clk-t(0)<6 d_Door.clk-Sys.clk<=0
d_Window.clk-Sys.clk<=0

Transition: d_Door._id6 -> d_Door.fired {clk > min
&& !system_done; a!; 1;} d_EnterRoom._id2 ->
d_EnterRoom._id1 {1; b?; 1;}

State: Sys.busy d_Door.fired d_Window._id3
d_EnterRoom._id1 f = 0 t = 1 system_done = 0
d_Door.min = 5 d_Door.max = 6 d_Door.id = 0
d_Window.min = 10 d_Window.max = 12 d_Window.id =
1 t(0)-Sys.clk<=0 t(0)-d_Door.clk<-5
t(0)-d_Window.clk<=0 Sys.clk-d_Window.clk<=0 h
d_Door.clk-t(0)<6 d_Door.clk-Sys.clk<=0
d_Window.clk-Sys.clk<=0

Transition: d_EnterRoom._id1 -> d_EnterRoom.done
{1; a!; 1;} Sys.busy -> Sys.done {1; a?;
system_done = 1;}

State: Sys.done d_Door.fired d_Window._id3
d_EnterRoom.done f = 0 t = 1 system_done = 1
d_Door.min = 5 d_Door.max = 6 d_Door.id = 0
d_Window.min = 10 d_Window.max = 12 d_Window.id =
1 t(0)-Sys.clk<=0 t(0)-d_Door.clk<-5
t(0)-d_Window.clk<=0 Sys.clk-d_Window.clk<=0
d_Door.clk-Sys.clk<=0 d_Window.clk-Sys.clk<=0
```