Universiteit Leiden
The Netherlands

Opleiding Informatica

`ADT Create`: Determination of Equivalence and Satisfiability of Attack-Defense Trees

Jayme Hebinck

Supervisors:
Nathan Daniel Schiele & Dr. Olga Gadyatskaya

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/07/2024

Abstract

Attack-Defense Trees (ADTs) are an important tool for visualizing security scenarios. The trees give a clear overview of a situation and showcase possible actions to take for both the proponent and the opponent. On the formal model of ADTs, Attack-Defense Terms (ADTerms), various mathematical operations can be performed. In this research, we focus on determining equivalence between two ADTerms and on determining satisfiability of an ADT. We introduce an equivalence threshold using the Levenshtein Distance to overcome the limitation of strict equivalence between two ADTerms. We also adopt the Depth-First Search (DFS) algorithm and apply it to ADTs to determine satisfiability of an ADT more efficiently than using the propositional semantics of ADTerms. We implement these operations in a tool that we develop, named `ADT Create`. We perform experiments to test the functionality of our methods, determine the optimal value of an equivalence threshold to overcome strict equivalence and determine how much more efficient our method for determining satisfiability is than using propositional semantics of ADTerms in determining satisfiability. We discuss the results and conclude our research with possible further research ideas.

Contents

1 Introduction

(Cyber)security is an important aspect of our society. More parts of our lives are being digitalized, and besides the fact that this makes our lives more convenient, it also makes us more vulnerable to cyber attacks, causing billions of dollars worth of damages [1]. In order to prevent and prepare ourselves for these attacks, we can use threat modeling to visualize these threats. Using threat modeling, we stop attacks by performing the correct counteractions, or even prevent attacks completely.

The threat modeling technique we focus on in this research is Attack-Defense Trees (ADTs), which have been introduced and formalized by Kordy et al. [2]. ADTs are a graphical representation of attack-defense scenarios described in a tree-like structure. The trees consist of multiple nodes, which can either have the attack or defense type. Together, these nodes represent a scenario, in which the goal is to either attack or defend a system. This goal is then split up in unique and detailed subgoals, which can be further split up in more subgoals. This way, an ADT gives a clear overview of a situation and showcases the possible actions that can be taken. ADTs make security scenarios easy to understand and prepares a user to stop or even fully prevent attacks against their system.

A formal representation of ADTs, called Attack-Defense Terms (ADTerms), is adopted from [2]. With this introduction of ADTerms, two models have been introduced to offer different capabilities for analyzing and reasoning about security scenarios. The multiset model introduces multiset semantics, which converts an ADTerm to a multiset representation. In this thesis, we look at multiset semantics for equivalence determination. The propositional model introduces propositional semantics, which converts an ADTerm to a propositional formula. In this thesis, we look at propositional semantics for satisfiability determination. We focus on these two operations and improve them.

Equivalence determination between two ADTerms using multiset semantics is done by processing the two ADTerms into two distinct multisets. In this process, the basic actions, which are the leaf nodes, processed in the two ADTerms are compared to each other. The two leaf node labels are only seen as equivalent, if they are exactly equal. This is a very strict approach, since the slightest differences between two labels lead to inequivalence between the two ADTerms, even if they describe the same scenario. This strictness therefore forms a limitation. This limitation can be overcome by allowing a certain amount of differences between two labels.

Satisfiability determination of an ADTerm using propositional semantics is done by converting an ADTerm into a propositional formula. This formula is then fully processed to determine if it is satisfiable. This process can become complex when the ADTerm grows in size, since satisfiability needs to be determined of every basic action and associated function symbol (intermediate nodes in an ADT). This process can be made more efficient by using ADTs, instead of ADTerms. Using ADTs, we can prune parts of the ADT that are not needed to determine satisfiability.

The improvements of these two operations are implemented in a tool that we develop, named `ADT Create`, together with all necessary features to upload, download, create and customize ADTs, and to generate ADTerms based on ADTs. This enables users worldwide to use an OS-independent tool to create and customize ADTs and use the improved operations for their own threat modeling.

1.1 Research objectives

In this thesis, we aim to obtain two research objectives. We go over these objectives, and state how we approach them.

Our first research objective (**RO1**) is: **Finding an optimal value for an equivalence threshold using the Levenshtein Distance to overcome strict equivalence between two ADTerms.**

We approach this objective by adopting the method to compare two ADTerms to each other from [2]. We then introduce an equivalence threshold, using the Levenshtein Distance, to create a margin, in which an amount of characters between labels is allowed to be different. We perform experiments to validate our methods of generating an ADTerm based on an ADT and of comparison between two ADTerms using an equivalence threshold using the Levenshtein Distance. Finally, we perform an experiment with a dataset of ADTs, to determine the optimal value for the equivalence threshold. This optimal value is the value that allows the most amount of ADTerms of ADTs to be seen as equivalent, whilst still maintaining the integrity of the comparison, and therefore is the value that reduces the strictness of the equivalence most with the best possible result.

Our second research objective (**RO2**) is: **Determine satisfiability of an ADT more efficiently than using the propositional semantics of an ADTerm.**

We approach this objective by adopting the theory behind the satisfiability determination of an ADTerm using propositional semantics. We use this theory as the foundation to create a method for determining satisfiability of ADTs. For this method, we adopt the Depth-First Search (DFS) algorithm from [3] to traverse an ADT and explore the possibilities of pruning parts of an ADT that are not needed to determine satisfiability. We perform experiments to validate our method and compare it to the process of determining satisfiability using the propositional semantics of an ADTerm to determine its efficiency.

1.2 Main contributions

The main contributions of this thesis are to introduce an equivalence threshold using the Levenshtein Distance and determine the optimal value for this margin to overcome the limitation of strict equivalence between two ADTerms. We also introduce a more efficient approach to determine satisfiability of an ADT using DFS, which is more efficient than using the propositional semantics of an ADTerm to determine satisfiability. We implement these optimized validated methods, together with the process of uploading, downloading, creating and customizing ADTs, in a tool that we develop, named `ADT Create`.

1.3 Thesis overview

The current chapter is the introduction of this thesis. Chapter 2 gives a detailed background of all topics that are discussed. This includes the generation of ADTerms based on ADTs, the comparison between two ADTerms using the multiset model and the determination of satisfiability of an ADTerm using the propositional model. For full understanding of the background, examples of all topics have been provided in Appendix A. Chapter 3 describes the related work that has been done prior to this thesis on the discussed background. In Chapter 4, we explain our proposed approach. This includes our proposed methods, the experiment designs we use for the experiments to test their functionality and efficiency to validate them, and the validity of the experiment designs. In Chapter 5, we go over the structure and implementation of our tool `ADT Create`. Chapter 6 goes over the results of the performed experiments. In Chapter 7, we discuss our research and use the results of the experiments to address our research objectives. Using the information gathered in our research, we then conclude this thesis and discuss potential further research in Chapter 8.

This bachelor thesis is supervised by Nathan Daniel Schiele and Dr. Olga Gadyatskaya and is part of the Leiden Institute of Advance Computer Science (LIACS).

2 Background

This chapter contains a detailed explanation of the background that is used in our research. We discuss the construction of ADTs, the generation of ADTerms, the comparison of ADTerms using multiset semantics, and the determination of satisfiability of an ADTerm using propositional semantics. Examples for all of these topics can be found in Appendix A.

2.1 Attack-Defense Trees

ADTs were first introduced and formalized in [2]. They are a graphical model which describes the actions an attacker or defender can take in order to achieve their goal of offending or defending a system [4]. ADTs consist of nodes. These nodes can have one out of two types: attack or defense. Attack nodes are generally displayed as red nodes and defense nodes as green nodes. A node N in an ADT can have children nodes. In this case, node N is a parent of these nodes, which makes it an intermediate node. Intermediate nodes form subgoals of the ADT. The children nodes of node N can have one out two possible relationships together. We refer to these relationships as the refinement between the children nodes. When a node N has the OR refinement, only one of its children nodes must be achieved to complete the subgoal the parent node represents. When a node N has the AND refinement, all of its children nodes must be achieved in order to complete the subgoal the parent node represents. If node N does not have children nodes, it is a leaf node, also referred to as a non-refined node. Leaf nodes are the basic actions that can be taken by an attacker or defender. Depending on the refinement, when a basic action, or combination of basic actions, is performed, the goal of the parent node can be achieved. When the goal of a parent node is achieved, or a combination of goals of parent nodes, the parent node of these parent nodes can be achieved. This continues, until we reach the final parent of the ADT, which does not have a parent node itself. We refer to this as the root node, which represents the main goal of the ADT.

ADTs have the possibility to process attributes [4], such as the cost to take basic actions and reach (sub)goals and the time that these basic actions and reaching these (sub)goals take. We do not process attributes in our research.

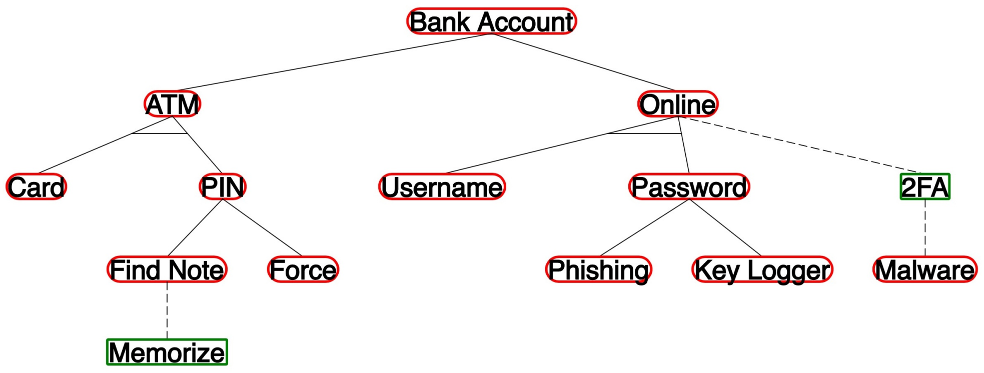


Figure 1: ADT depicting an attack on a Bank Account. Modified version of the ADT made by Kordy et al. in [2]. Made using the ADT WebApp [5].

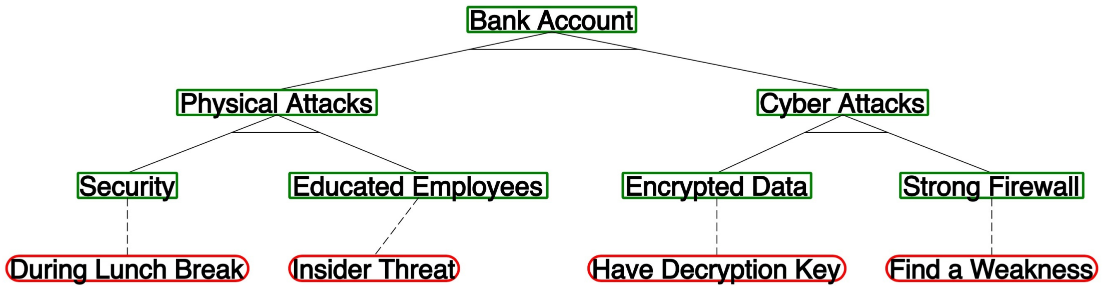


Figure 2: ADT depicting a defense for a Bank Account. Made using the ADT WebApp [5].

The descriptions of the ADTs in Figure 1 and in Figure 2 can be found in Appendix A.1 and Appendix A.2 respectively.

2.1.1 Countermeasure Prevention

In order for an ADT to be as effective as possible, certain rules have been set into place. We refer to these rules as countermeasure prevention. Countermeasure prevention states that nodes of a certain type may have at most one child node that has the opposite type [4]. This means that an attack node may have at most one defense child node, and a defense node may have at most one attack child node. Because of this rule, nodes describe a unique scenario for which there may be at most one counter. This counter may be broad, and form its own subADT describing this counter in more detail. In an ADT, the connection between a parent node and a counter child node is given with a dotted line, instead of a straight line.

2.2 ADTerms

We adopt a formal representation of ADTs, namely ADTerms, from [2]. ADTerms have certain rules set in place for its notation. We refer to this notation as the AD-signature [2]. This signature consists of two sets. The first set represents the distinction between the two types attack and defense. However, since the root node decides which of the two types is the main goal of the scenario, we refer to the root type as the proponent and to the other type as the opponent. The second set contains function symbols, which represent the possible relationships between parts of an ADTerm.

Formally, the AD-signature [2] is a pair $\Sigma = (\mathcal{S}, \mathcal{F})$, where

- $\mathcal{S} = \{\mathrm{p}, \mathrm{o}\}$ is a set of the two types, representing the proponent and opponent, and
- $\mathcal{F} = \mathbb{B}^{\mathrm{p}} \cup \mathbb{B}^{\mathrm{o}} \cup \{\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}, \vee^{\mathrm{o}}, \wedge^{\mathrm{o}}, \mathrm{c}^{\mathrm{p}}, \mathrm{c}^{\mathrm{o}}\}$ is a set of function symbols. $\mathbb{B}^{\mathrm{p}}$ and $\mathbb{B}^{\mathrm{o}}$ define the basic actions that the proponent and opponent can take. $\{\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}, \vee^{\mathrm{o}}, \wedge^{\mathrm{o}}, \mathrm{c}^{\mathrm{p}}, \mathrm{c}^{\mathrm{o}}\}$ defines a set of functions that describe the relationship, also referred to as refinement, between the children nodes of proponent and opponent nodes.

Using this AD-signature, we can build up terms. These terms we call ADTerms, and represent ADTs.

In the following sections, we go over the different models for ADTerms and the generation of ADTerms based on ADTs.

2.2.1 Models for ADTerms

Kordy et al. introduce and define two models for ADTerms in [2], namely the propositional model and the multiset model, which they extended from Attack Trees (ATs). In this section, we go over these models. In the following two sections, we refer back to these models and reason which model we use for our research.

Propositional model The propositional model is based on propositional logic [6]. An ADTerm is converted to a propositional formula by associating a propositional variable to each basic action. An ADTerm is satisfiable when there is a combination of truth values of the basic actions, for which the final computed value of the root node holds. Two ADTerms in the propositional model are equivalent iff for every valuation υ of the propositional variables in the two propositional formulae ψ and ψ' holds that $\upsilon(\psi) = \upsilon(\psi')$. The semantics induced by this model are called propositional semantics. The formal definition of the propositional model can be found in [2].

Multiset model The multiset model is based on multisets. A multiset is an unordered collection of elements in which an element can appear more than once [7]. In the context of ADTerms, the elements are the basic actions. Bossuat and Kordy have made a distinction between two types of duplicate nodes, namely cloned nodes and twin nodes, in [8]. For our research, we only acknowledge twin nodes, which are duplicate nodes that are completely independent of each other. Multisets are a suitable manner of storage for ADTerms, since basic actions are mandated by AND and OR refinements, which are commutative. Therefore, the order of the basic actions does not matter, which is one of the key features of multisets. An ADTerm is satisfiable when there is a selection of elements that the proponent can achieve to achieve the subgoals that are necessary to achieve the main goal. Attributes can also be evaluated in satisfiability determination using this model, since multiple possible selections of elements and their attribute values can be compared to each other. Two ADTerms in the multiset model are equivalent if the multiset of both ADTerms contain the same elements, including twin nodes and in any order. The semantics induced by this model are called multiset semantics. The formal definition of the multiset model can be found in [2].

The multiset model is proved to be finer than the propositional model in [2], since the propositional model does not distinguish multiple occurrences of a basic action, whilst the multiset model does. Therefore, when two ADTerms with twin nodes are being compared using the propositional model, they may be seen as equivalent, due to this model not taking into account multiple occurrences of a basic action, whilst it is not seen as equivalent using the multiset model. With our two research objectives in mind, we look into the propositional model for determining satisfiability and the multiset model for determining equivalence.

2.2.2 Generation of ADTerms

ADTerms are generated based on ADTs. We determine structures of ADTs, defined by Kordy et al. [2], by looking at the ADTs and determine relationships between nodes following the AD-signature introduced in Section 2.2. This process is simplified by the introduction of a transformation table, which can be seen in Table 1 and is adopted from [2]. This table shows the transformation rules for all possible ADT structures to ADTerms. Using these eight transformation rules, any

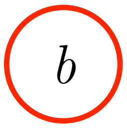
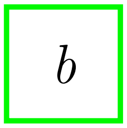
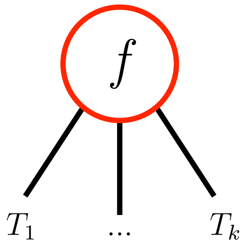
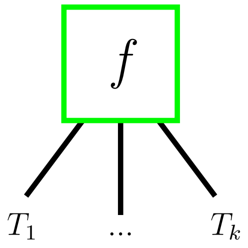
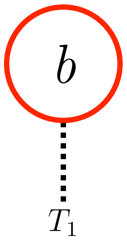
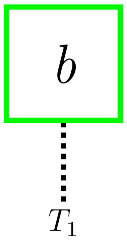
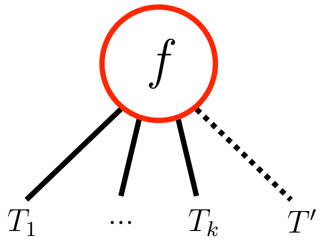
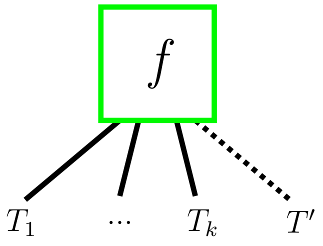
	Cell 1	*Cell 2*	*Cell 3*	*Cell 4*
T	 where $b \in \mathbb{B}^{\text{p}}$	 where $b \in \mathbb{B}^{\text{o}}$	 where $f \in \{\vee^{\text{p}}, \wedge^{\text{p}}\}, k \geq 1$	 where $f \in \{\vee^{\text{o}}, \wedge^{\text{o}}\}, k \geq 1$
$\iota(T)$	b	b	$f(\iota(T_1), ..., \iota(T_k))$	$f(\iota(T_1), ..., \iota(T_k))$
	Cell 5	*Cell 6*	*Cell 7*	*Cell 8*
T	 where $b \in \mathbb{B}^{\text{p}}$	 where $b \in \mathbb{B}^{\text{o}}$	 where $f \in \{\vee^{\text{p}}, \wedge^{\text{p}}\}, k \geq 1$	 where $f \in \{\vee^{\text{o}}, \wedge^{\text{o}}\}, k \geq 1$
$\iota(T)$	$\text{c}^{\text{p}}(b, \iota(T_1))$	$\text{c}^{\text{o}}(b, \iota(T_1))$	$\text{c}^{\text{p}}(f(\iota(T_1), ..., \iota(T_k)), \iota(T'))$	$\text{c}^{\text{o}}(f(\iota(T_1), ..., \iota(T_k)), \iota(T'))$

Table 1: Transformation table for transformation from ADTs to ADTerms [2].

ADT T can be transformed into ADTerms $\iota(T)$. What is important to note, is that in ADTerms, only the labels of the basic actions are present. The labels of intermediate nodes do not matter, since the computations and internal structure of ADTs do not depend on them. Therefore, they are only represented by their associated function symbols which can be found in the AD-signature [2].

To generate ADTerms, we adopt the method by Kordy et al. [2]. We work bottom-up, and start at the deepest level of abstraction of the ADT, where we determine the first basic actions. We then keep working levels up and define the relationship between basic actions, intermediate nodes and parts of the already generated ADTerms using the transformation table in Table 1, until we reach the root node. The ADTerms is then fully generated.

An example of the generation of an ADTerm can be found in Appendix A.3.

2.3 Equivalence between ADTerms

In an ADT, the OR and AND refinement of a node determine the relationship between the children nodes. These refinements are commutative, meaning that the order of the children nodes does not affect the meaning of the ADT. We determine equivalence between two ADTerms using the multiset model, since, when determining equivalence, we are interested in the finest level of detail, and therefore want to be able to distinguish twin nodes. Twin nodes are further discussed in Section 2.2.1. We say that two ADTs are equivalent if they represent the same multiset representation generated using the multiset semantics.

To determine equivalence using multiset semantics, we convert two ADTerms to two multisets using the multiset model by Kordy et al. [2]. We then compare these two multisets to each other. We determine equivalence between these two multisets by checking whether the multisets contain the same elements, including twin nodes, and in any order. An example of determining equivalence using multiset semantics can be found in Appendix A.4. In order for these two multisets to be equivalent, the labels of the basic actions must be equal to each other. This is a very strict approach. Therefore, we look into creating an equivalence threshold using the Levenshtein Distance. The threshold represents a percentage of the amount of characters of the longest label in the comparison that is allowed to differ between the two strings.

Levenshtein Distance In order to create an equivalence threshold, we adopt the Levenshtein Distance from [9]. The Levenshtein Distance determines the minimal amount of operations needed to change one string into another. These operations are insertions, deletions and substitutions [10]. Using this method, we can determine how many operations two strings, in our case the labels of two basic actions, are apart from each other, and allow an amount of characters to differ. An example of using the Levenshtein Distance between two strings to determine the amount of operations that is needed to change one string into the other, can be found in Appendix A.5.

2.4 Satisfiability of an ADTerm

In order to determine satisfiability, we discuss the model that is most relevant to our research. As can be read in Section 2.2.1, there are important distinctions between the propositional model and the multiset model. The multiset model adds the comparison of attributes in ADTs to determine the best satisfiable path, instead of just a satisfiable path. However, since in our research, we are not interested in processing attributes in the determination of satisfiability, as is stated in Section 2.2.1, we choose to work with the propositional model. This model covers determining satisfiability of the structure of an ADTerm, and therefore of an ADT, which is what we are interested in in our research.

Using the described notation from the propositional model [2], we can determine the propositional semantics of an ADTerm, with which we can determine if the ADTerm is satisfiable. To ensure the results of our research are consistent, we assume we can perform all basic actions (attack and defense basic actions) that are present in the ADTs, and give them a value 1. This does not bias the research, since we assume that all basic actions are satisfiable in both our method and determining satisfiability using propositional semantics, which we compare to each other to determine the efficiency of our method. Two examples of determining satisfiability of an ADTerm using propositional semantics, one of which satisfiable and the other not satisfiable, where all basic actions are assumed to be performable, can be found in Appendix A.6 and Appendix A.7 respectively.

Satisfiability in an ADT, where all basic actions are assumed satisfiable, means that there must be at least one path made out of proponent nodes, that is not countered by opponent nodes. This decides whether or not the proponent here is the overall winner of the ADT. However, if every possible proponent path is countered by an opponent node, the opponent is the overall winner, making the ADT not satisfiable [4].

3 Related Work

This chapter discusses the related work that has been done prior to this thesis. We discuss multiple papers that have been written about similar topics and show their relevance to our research. Specifically, we discuss the research done on ADTs, the tools that have been developed to create and represent ADTs and the implementation of determination of equivalence and satisfiability in other areas.

3.1 History of ADTs

ADTs form the core of our research. They are an extended version of Attack Trees (ATs), which were introduced in 1999 by Schneier [11] and later formalized by Mauw and Oostdijk in 2006 [12]. In 2011, ADTs were formalized by Kordy et al. stating the foundations of ADTs by introducing defense nodes in ATs and a formal representation of ADTs: ADTerms [2]. The generation of ADTerms, the process of determining equivalence and satisfiability of ADTs and countermeasure prevention are also introduced in this formalization of ADTs. It also introduces the propositional model and multiset model for ADTs, which both also have been extended from ATs [13].

Shortly after in 2011, Kordy et al. proved that the introduction of defense nodes in ATs does not increase the computational complexity of ADTs [14], showing the efficiency of ADTs. In 2012, Kordy, et al. continued their research on ADTs and published a paper which guides a user in formulating quantitative questions on ADTs and how to construct corresponding attribute domains for them [4]. In this same year, Kordy et al. expanded upon the foundations of ADTs by giving complete axiomatizations for the propositional and multiset semantics of ADTs [15].

3.2 Tools

To address our research objectives, a tool is developed that contains all functionalities needed. Similar created tools include ADTool [16], which is a tool for creating and representing ADTs in a visual appealing manner, based on the formal framework of ADTs [2]. It supports the conversion of ADTs to ADTerms and allows the connection of attributes to the nodes. It is specifically created to guide users in constructing correct trees, is available for free and open source. ADTool is based on other tree-like modeling tools, such as SecurITree [17] and AttackTree+ [18], which are similar tools behind a paywall and are closed source. Other tools, such as SeaMonster, are open source and are free, but do not have the same capabilities as ADTool [16, 19]. For example, SeaMonster is only able to construct ATs and Misuse Cases, and not ADTs. However, ADTool cannot determine equivalence between two ADTerms. Besides that, the tool is only available for the Windows operating system, meaning it is not universally available. An app that is universally available, is the ADT WebApp [5]. It misses out on some functionalities that ADTool has, like the support for attributes and ADTerms, but is available as a web application on any webbrowser and is currently still under development. All figures of ADTs in this thesis are created using the ADT WebApp.

3.3 Equivalence

We are determining equivalence between two ADTerms, to see if they represent the same scenario, even though basic actions might slightly differ in order. Research for equivalence between other types of trees has been done as well. Zantema has researched the equivalence between decision trees [20], which have a similar hierarchical structure as ADTs. Another example is fault trees [21], for which equivalence between two fault trees can also be determined in a similar way due to a similar hierarchical structure.

An important part of how we determine equivalence between similar scenarios includes determining if the labels of the basic actions of two ADTerms are the same or not. However, direct comparison can determine inequivalence between two ADTerms, even if they describe the same scenario. Therefore, we use methods to determine similarity between two basic actions. Effectively, we are doing string comparisons here. String comparison is a thoroughly researched area. Other than the Levenshtein Distance [10], which we use in this thesis, there are also other string comparison methods. One of them is the Hamming Distance [22], which calculates the number of positions in which the characters are different between two strings. A downside of this method is the need of two strings with the exact same length, which is unlikely for the labels of two basic actions. Another set of methods for string comparison is referred to as phonetic algorithms [23]. A phonetic algorithm takes a word and transforms it into a phonetic code. This code indicates the pronunciation of the word in a specific language, which is then matched with other words with similar pronunciations. An example of a phonetic algorithm is Soundex [24], which is being researched for its use in SMS text representation.

3.4 Satisfiability

The term satisfiability origins from the field of propositional logic [6]. In our case, the satisfiability problem is applied to ADTs to determine whether the proponent can defeat the opponent in their attack-defense scenario or not. However, it has been applied in other areas as well. Some of these areas include, but are not limited to, combinational equivalence checking, automatic test-pattern generation, model checking, planning and haplotype inference [25]. Even though the satisfiability problem is a NP-complete decision problem [26], the algorithm is remarkably efficient in these areas.

4 Proposed Approach

In this chapter, we introduce our proposed approach by going over our proposed methods and how they together form the approach to our research objectives. We also go over the experiment designs of our experiments to test the functionality and effectiveness of our methods and the validity of the experiment designs.

4.1 Proposed Methods

This section contains the explanations of our proposed methods that we use to achieve our research objectives. We discuss the algorithms for the generation of ADTerms, the comparison of ADTerms and the determination of satisfiability using DFS on ADTs.

4.1.1 Method for Generation of ADTerms

The generation of an ADTerm is started from the root node and performed recursively. We start with generating an empty string, and build up the ADTerm bottom-up. From the root node, we go through the ADT using DFS. We start with generating parts of the ADTerm based on the basic actions. Using these parts, we can build up parts of the ADTerm based on intermediate nodes, since these parts of the ADTerm are dependent on the children nodes. We continue this process, and merge the parts of an ADTerm together, up until the root node. To generate an ADTerm, we use the transformation table which can be found in Table 1.

Algorithm 1 shows the algorithm for ADTerm generation of an ADT. The algorithm, together with the helperfunctions, represent all the possible conversion possibilities of the base structures in Table 1. We state which parts of the algorithm generate the ADTerm of which base structures in Table 1. We do this by referring back to the cell numbers in the table. The algorithm combines the generated parts of the ADTerm. It also generates the ADTerm of a leaf node (cell 1 and cell 2), and generates the ADTerm of an intermediate node with a single child node that are both of the same type (cell 3 and cell 4 for $k == 1$). `ADTermSingleChildWithCounter(node` N`, result)` in Listing 1 generates the ADTerm of a node with a single child node, which is a counter child node (cell 5 and cell 6). `ADTermIntermediateWithCounter(node` N`, result)` in Listing 2 generates the ADTerm of an intermediate node, meaning it has other children nodes of the same type, with a counter child node (cell 7 and cell 8). Lastly, `ADTermIntermediate(node` N`, result)` in Listing 3 generates the ADTerm of an intermediate node with more than one child node and no counter child node (cell 3 and cell 4 for $k > 1$).

The notation of the resulting ADTerm might look slightly different from the notation by Kordy et al. in [2], due to the limitations of support for subscript and superscript characters in Bash with a focus on clean and readable code. This difference of notation does not negatively impact the representation of the ADTerm. The algorithm has a computational complexity of $\mathcal{O}(n)$, in which n is the total amount of nodes in the ADT. We use this method to generate ADTerms based on ADTs in order to compare these ADTerms to determine equivalence.

Algorithm 1: Generation of ADTerm of a given ADT, based on the theory by Kordy et al. [2]

```
Input: Node $N$
Output: ADTerm of node $N$ in string format
Initialize result = ””
foreach child $C$ of node $N$ do
    # Go through ADT recursively
    Generate ADTerm of child $C$
    if first child $C$ of node $N$ then
        result += ADTerm of child $C$
    else
        result += ’, ’ + ADTerm of child $C$
if node $N$ is a leaf node then
    result = label node $N$ # Leaf node, ADTerm is label
else
    # node $N$ is not a leaf node
    if node $N$ has one child $C$ then
        if type child $C$ == type node $N$ then
            # No counter child node
            result = ’[’ + result + ’]’ # No function symbol necessary, indicate deeper
              level of abstraction
        else
            # Has counter child node
            result = ADTermSingleChildWithCounter(node $N$, result) # See Listing 1
    else
        # Node $N$ has more than one child node
        if node $N$ has a counter child node then
            result = ADTermIntermediateWithCounter(node $N$, result) # See Listing 2
        else
            # Node $N$ does not have a counter child node
            result = ADTermIntermediate(node $N$, result) # See Listing 3
return result
```

```
def ADTermSingleChildWithCounter(node N, result):
  if type node N == type root node ADT:
    result = 'cp[' + label node N + ', ' + result + ']' # Proponent, has counter child
  else:
    result = 'co[' + label node N + ', ' + result + ']' # Opponent, has counter child
  return result
```

Listing 1: Pseudocode for function for generating the ADTerm of an ADT node N with a single child node, which is a counter child node.

```
def ADTermIntermediateWithCounter(node N, result):
  functionSymbol = ''
  if type node N == type root node ADT:
    if node N has OR refinement:
      functionSymbol = 'cp[∨p[' # Proponent, OR refinement, has counter child
    elif node N has AND refinement:
      functionSymbol = 'cp[∧p[' # Proponent, AND refinement, has counter child
  else:
    if node N has OR refinement:
      functionSymbol = 'co[∨o[' # Opponent, OR refinement, has counter child
    elif node N has AND refinement:
      functionSymbol = 'co[∧o[' # Opponent, AND refinement, has counter child
  # Split counter child node from other children nodes of node N
  ADTermChildren = ADTerm of children of same type as node N from result
  ADTermCounter = ADTerm of counter child node of node N from result
  result = functionSymbol + ADTermChildren + '], ' + ADTermCounter + ']'
  return result
```

Listing 2: Pseudocode for function for generating the ADTerm of a node N with multiple children of the same type and a counter child node.

```
def ADTermIntermediate(node N, result):
  if type node N == type root node ADT:
    if node N has OR refinement:
      result = '∨p[' + result + ']' # Proponent, OR refinement
    elif node N has AND refinement:
      result = '∧p[' + result + ']' # Proponent, AND refinement
  else:
    if node N has OR refinement:
      result = '∨o[' + result + ']' # Opponent, OR refinement
    elif node N has AND refinement:
      result = '∧o[' + result + ']' # Opponent, AND refinement
```

Listing 3: Pseudocode for function for generating the ADTerm of a node N with multiple children of the same type and no counter child node.

4.1.2 Method for Comparison between ADTerms

In the process of determining equivalence between two ADTerms, we do not use the multiset semantics. The reason for this is the efficiency of our method. In our method, we can stop the comparison process immediately when it is determined that two ADTerms are not equivalent. This is not possible when using the multiset model, due to the fact that the entire multiset representation of both ADTerms need to be generated, and need to be fully compared to determine equivalence or inequivalence. This does mean that our method cannot determine exact equivalence between two ADTerms, since we do not use the multiset semantics. Therefore, we state that our method can determine a restricted form of equivalence. This restricted form is suitable for the experiment to optimize the equivalence threshold, which we perform on our method. In the next paragraphs, we explain our modified approach and the implementation of the equivalence threshold in this approach. This approach does use multisets, and allows us to distinguish twin nodes, which are explained in Section 2.2.1.

A comparison happens in two methods, the top-down method and the bottom-up method. The top-down method checks for every level of abstraction, if the same associated function symbols, basic actions and multisets are present. The bottom-up method checks if the parent-child relationships between the elements and their associated function symbols in the two ADTerms are the same. This is necessary to determine, to know if an element falls in the scope of the same parent associated function symbol in both ADTerms.

In Algorithm 2, you can see the pseudocode of the ADTerms comparison process to determine equivalence. The input is two ADTerms $t1$ and $t2$ in string format, and the output is a boolean value indicating equivalence and an int value representing the total Levenshtein Distance, if the ADTerms are equivalent within the equivalence threshold. We first check if the ADTerms both start with an associated function symbol or not. This would mean that the root node has more than one child node. If this is the case, we check if these symbols are the same, and continue if this is the case. Otherwise, the comparison is immediately stopped, and inequivalence is returned to the user. We then start our recursive comparison, initiating with the top-down method, where we extract the elements located a level deeper in the ADTerms, which represent the children nodes of the root node. We give the parameter *equivalence* the value `True`; we assume equivalence, until it is determined that these ADTerms are not equivalent. We process the elements of the ADTerms in two distinct multisets to determine if they are equal, and if the Levenshtein Distance between the leaf nodes are within the equivalence threshold, which happens in `compareMultisets`$(t1, t2)$ in Listing 4. If this is true, we recursively go a level deeper into the ADTerms. We continue this process, until we determined that every level of the two ADTerms contain the same elements. If at any point, inequivalence is determined, the comparison is stopped, and inequivalence is returned to the user. Then, the bottom-up method starts in the recursive comparison. In the deepest level of the ADTerms (which are the leaf nodes), we connect the leaf nodes to their associated function symbol a level higher. We do not have to check the connections between elements that are not leaf nodes and associated function symbols, since this is part of the structure, which is already determined to be equivalent in the top-down method. The connections are stored in pairs. These pairs of both ADTerms are then being compared, and if they are equal, we continue the comparison process. This process happens in `parentChildCheck`$(t1, t2, t'1, t'2)$ in Listing 5. If they are not equal, we stop the comparison process, and inequivalence is returned to the user. This continues, until we are back in the root node. Equivalence is then returned out of recursion, and together with the total Levenshtein Distance that was needed in this comparison process, this is returned to the user. The algorithm has a computational complexity of $\mathcal{O}(n \times m)$, in which n is the total amount of nodes of the first ADT processed into the first ADTerm, and m is the total amount of nodes of the second ADT processed into the second ADTerm. We use this method to compare two ADTerms in order to determine equivalence between them. The equivalence threshold is used to make this comparison process less strict. We use this method to test for various values for the equivalence threshold, in order to achieve **RO1**.

Algorithm 2: Equivalence determination between two ADTerms (in string format) including an equivalence threshold, using the top-down method and bottom-up method.

```
Input: ADTerm $t1$ and ADTerm $t2$, both in string format
Output: Boolean value indicating equivalence, int value representing total Levenshtein
        Distance (if the two ADTerms are not equivalent, this value is 0)
Initialize totalLevenshteinDistance = 0
if t1 and t2 both start with a function symbol or leaf node then
    # Initial check before recursive comparison
    initialResult = compareMultisets(t1, t2) # See Listing 4
    if initialResult determines inequivalence then
        return False, 0
    else if t1 and t2 start with a leaf node, which is equivalent then
        totalLevenshteinDistance += Levenshtein Distance between the two leaf nodes
equivalence = True # Assume equivalence, determine inequivalence in the process
start the recursive comparison ← Recursive calls start here, $t'1$ and $t'2$ become $t1$ and $t2$
    start top-down method
        if equivalence then
            $t'1$, $t'2$ = Extract elements from associated function symbols of $t1$ and $t2$, discard
              current level elements # Go a level deeper in the ADTerms
            if $t'1$ and $t'2$ are both empty strings then
                # Empty strings means top-down method is completed
                Leave level of recursion, with True , totalLevenshteinDistance, t'1, t'2
            compareMSResult = compareMultisets($t'1$, $t'2$) # See Listing 4
            if compareMSResult determines inequivalence on this level of abstraction then
                if not in the most outer level of the ADTerm yet then
                    Leave level of recursion, with False, 0, t'1, t'2
                else
                    End recursive comparison with equivalence, totalLevenshteinDistance
            totalLevenshteinDistance += Levenshtein Distance calculated in compareMSResult
    equivalence, totalLevenshteinDistance, t"1, t"2 = Recursive comparison with t'1, t'2
    start bottom-up method
        if equivalence then
            parentChildResult = parentChildCheck($t'1$, $t'2$, $t''1$, $t''2$) # See Listing 5
            if parent-child relationships are not the same in the two ADTerms then
                equivalence = False
                totalLevenshteinDistance = 0
        if not in the most outer level of the ADTerm yet then
            Leave level of recursion, with equivalence, totalLevenshteinDistance, $t'1$, $t'2$
        else
            End recursive comparison with equivalence, totalLevenshteinDistance
return equivalence, totalLevenshteinDistance
```

```
def compareMultisets($t1, t2$):
  # ADTerm in string format, convert to multiset
  $MSt1, MSt2$ = Convert elements from current level of abstraction in
  string format ADTerms to multisets containing these elements
  partLevenshteinDistance = 0
  for each element in $MSt1$:
    if element is a leaf node:
      # Determine if two leaf nodes are within equivalence threshold
      if element also in $MSt2$ (within equivalence threshold):
        partLevenshteinDistance += Levenshtein Distance between these two
        leaf nodes
        Replace equivalent leaf node in $MSt2$ with element from $MSt1$
        # To allow for multiset comparison, elements must be equal, so
        # temporarily change one of the two elements to the other. $MSt1$
        # and $MSt2$ are discarded after this comparison is finished, and
        # therefore this change does not have any influence on the the
        # rest of the comparison process.
  if $MSt1$ and $MSt2$ contain the same elements:
    return True, partLevenshteinDistance
  else:
    return False, 0
```

Listing 4: Pseudocode for function for determining equivalence between two multisets (leaf nodes within equivalence threshold), containing the elements of a level of abstraction of an ADT (top-down method).

```
def parentChildCheck($t1, t2, t'1, t'2$):
  Convert $t1, t2, t'1, t'2$ into multisets
  Pairs $P1$ = Connect leaf nodes from multiset $t'1$ to associated function symbols
  from multiset $t1$
  Pairs $P2$ = Connect leaf nodes from multiset $t'2$ to associated function symbols
  from multiset $t2$
  if $P1$ and $P2$ contain the same pairs (leaf nodes within equivalence threshold):
    return True
  else:
    return False
```

Listing 5: Pseudocode for function for checking the parent-child relationships between the leaf nodes (within equivalence threhsold) and their associated function symbols (bottom-up method).

Algorithm 3: Determining Satisfiability using Recursive DFS applied on ADTs.

```
Input: Node N
Output: Boolean value indicating whether node N is satisfiable
1  if node N has > 1 child then
2      if node N has OR refinement then
3          foreach child C of node N do
4              Determine satisfiability of child C
5              if type node N == type child C and child C is satisfiable then
                   # Satisfiable child C found
6                  foreach child C of node N do
7                      if type node N != type child C then
8                          Determine satisfiability of child C
9                          if child C is satisfiable then
10                             return False # Satisfiable counter child C is found, so
                                 node N is not satisfiable
11                 return True # Satisfiable child C found, no satisfiable counter
                     child found, so node N is satisfiable
12             else if type node N != type child C and child C is not satisfiable then
13                 return True # Not satisfiable counter child C found, node N is
                     satisfiable
14         return False # No satisfiable child C found, node N is not satisfiable
15     else if node N has AND refinement then
16         foreach child C of node N do
17             Determine satisfiability of child C
18             if type node N != type child C and child C is satisfiable or
19             type node N == type child C and child C is not satisfiable then
20                 return False # Satisfiable counter child or not satisfiable child
                     found, so node N is not satisfiable
21         return True # All children are satisfiable, except for a possible counter
             child, so node N is satisfiable
22 else if node N has 1 child C then
23     Determine satisfiability of child C
24     if type node N == type child C and child C is satisfiable or
25     type node N != type child C and child C is not satisfiable then
26         return True # Child C is satisfiable, or counter child C is not
             satisfiable, so node N is satisfiable
27     return False # Child C is not satisfiable, or counter child C is
         satisfiable, so node N is not satisfiable
28 else if node N is a leaf node then
29     return True # Assume leaf nodes are satisfiable
```

4.1.3 Method for Satisfiability

By determining satisfiability of an ADT, we know whether the proponent reaches their goal, or if the opponent has enough counters in place to stop the proponent from reaching their goal. More information about the background behind satisfiability using the propositional semantics can be found in Section 2.4. The goal of our modified method is to be able to determine satisfiability more efficiently. We reach this goal by demonstrating we can determine satisfiability using fewer nodes than when using the propositional semantics of an ADTerm. In our modified method, we determine satisfiability of an ADT, instead of of its formal representation, an ADTerm. Using the DFS algorithm, we can prune parts of an ADT during the process. Satisfiability for ADTs with an offensive goal is determined in the same way as for ADTs with a defensive goal.

In Algorithm 3, you can see the pseudocode of the algorithm that is used to determine satisfiability. The input is a node N, and the output a boolean value indicating whether this node N is satisfiable. In lines 2-14, the process of determining satisfiability on a node with OR refinement is described. This node only needs one satisfiable path down the ADT in order to be satisfiable. Using DFS, we can check every child node, and when we find a satisfiable child node, and there is no satisfiable counter child node, we can prune the rest of the children nodes. In lines 15-21, the process of determining satisfiability on a node with AND refinement is described. This node needs all children nodes to be satisfiable, and a possible counter child node to be not satisfiable. If one child node is found to be not satisfiable, or a satisfiable counter child node is found, then we can prune the rest of the children nodes. These two possibilities to prune children nodes in an ADT can lead to large parts of ADTs that do not need to be considered, leading to a more efficient process than using an ADTerm of an ADT that needs to be fully processed in order to determine satisfiability. In our method, we assume all leaf nodes are satisfiable, due to the reasons provided in Section 2.4. The algorithm has a computational complexity of $\mathcal{O}(n)$, in which n is the total amount of nodes in the ADT. We use this method to determine satisfiability on ADTs, using DFS to explore the possibilities of pruning parts of the ADT. This method is used to compare against determining satisfiability using propositional semantics, to achieve **RO2**.

4.2 Experiment Designs

In this section, we discuss the design of our experiments to test the discussed methods in Section 4.1 for a reasonable amount of test cases. These tests are trivial, we expect them to succeed for the given base structures. We also discuss the design of our experiments to find the optimal value of an equivalence threshold and to test the efficiency of our method for determining satisfiability of an ADT in comparison to determining satisfiability using the propositional model, which is explained in Section 2.2.1.

4.2.1 Functional Experiments for Proposed Methods

We test the functionality of the generation of an ADTerm (Algorithm 1), the comparison of two ADTerms (Algorithm 2) and the determination of satisfiability (Algorithm 3), using a similar approach, namely by validating that the algorithms behave as supposed to for the given base structures in Table 1. We do this by creating examples that individually describe the given base structures. These examples can be found in the figures underneath. Examples with the defense type as proponent are processed in the same way as the examples we use with the attack type as proponent.

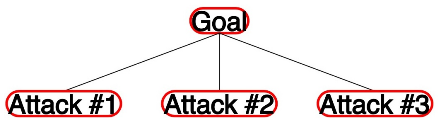


Figure 3: ADT Example with OR refinement and no counter child node. Made using the ADT WebApp [5].

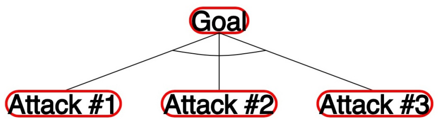


Figure 4: ADT Example with AND refinement and no counter child node. Made using the ADT WebApp [5].

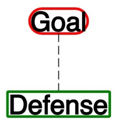


Figure 5: ADT Example with a single direct counter child node. Made using the ADT WebApp [5].

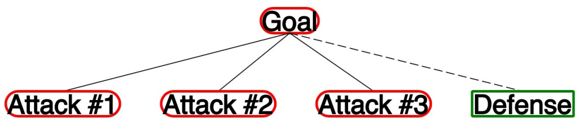


Figure 6: ADT Example with OR refinement and a direct counter child node. Made using the ADT WebApp [5].

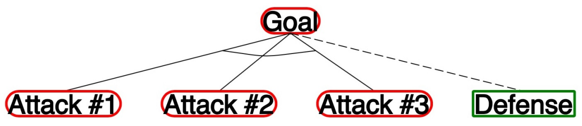


Figure 7: ADT Example with AND refinement and a direct counter child node. Made using the ADT WebApp [5].

We perform the experiments for determining functionality of ADTerms generation and satisfiability determination and compare them to the mathematical processes that are described in Chapter 2. This way, we show if the results of our methods give the same (relevant parts of the) results of the mathematical processes, demonstrating that our methods are valid for these base cases. These tests are trivial. We expect them to succeed to show that our methods work reasonably to perform our other two experiments that help us get to our research objectives, one helping to cover **RO1** and one covering **RO2**. These two experiments are explained in the following two sections.

The experiment for determining functionality of the comparison of ADTerms is done by comparing the ADTs in the figures above to themselves. By showing that comparing an ADTerm to itself yields a result of equivalence, we validate our algorithm, since the algorithm stops immediately when inequivalence is determined. Therefore, the full process of the algorithm can only be shown by comparing two equivalent ADTerms, which can be done by comparing an ADTerm to itself. Since this algorithm is not based on the multiset model, we cannot prove equivalence between two ADTerms. In our method, the structure of both ADTs, based on which the ADTerms are generated, must have a similar structure in order to be seen as equivalence; intermediate nodes are allowed to be in a different order, but all have to be present. In the multiset model, there are certain cases where the structure can deviate from each other. Therefore, this trivial experiment demonstrates that our method can determine a restricted form of equivalence, which is reasonable enough to perform our other experiment on that helps us get to our research objective **RO1**.

4.2.2 Equivalence Threshold Experiment

Determining equivalence between two ADTerms means that the labels of all the basic actions should be directly equal in both ADTerms. This can be very inefficient, since two labels can have a small difference, but still have an equal meaning. In order to have a small margin of error that is accepted as equivalent between two labels, we have adopted the Levenshtein Distance in Section 2.3. We show the implementation in Algorithm 2 in Section 4.1.2. In this experiment, we determine the optimal equivalence threshold using the Levenshtein Distance that should be allowed between the labels of two basic actions in order to yield the most correctly determined equivalent ADTerms and the least incorrectly determined non-equivalent ADTerms. We have created a dataset containing 24 ADTs trying to describe the same scenario. The scenario that the ADTs try to describe is:

You want to break into a safe. There are two ways you can do this. The first way is to gain access via the door. In order to do this, you would need to steal an access card and make sure the security is on lunch. However, the security has 2FA installed on the safe door as a safety measure. The second way is to gain access via the ventilation. This can be done by finding a vent opening.

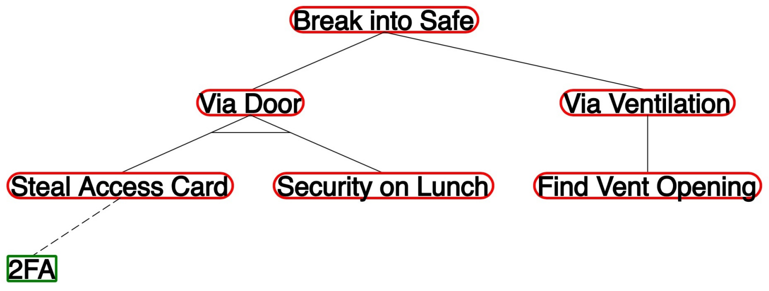


Figure 8: ADT that describes the given scenario most accurately. It is the control proof for the experiment to test the effectiveness of the Levenshtein Distance in determining equivalence between two ADTerms. Made using the ADT WebApp [5].

The ADT that describes the scenario most accurately is shown in Figure 8. We use this ADT to compare to all other ADTs; it is our control proof. ADTs 2-4 describe the scenario, each with one character difference in a label of a leaf node. ADT 2 changes the shortest label `"2FA"` by one character, ADT 3 changes the longest label `"Steal Access Card"` by one character and ADT 4 changes both by one character. This way, we process the effect of the Levenshtein Distance on shorter labels (three characters) and longer labels (seventeen characters). ADTs 5-7 are the same as ADTs 2-4, but with two character differences per label. From ADT 8 on, we focus ourselves on one specific control node, namely `"Steal Access Card"`. ADTs 8-18 will describe the scenario, but every next ADT will have one more character different in this label compared to this label in the control proof, starting with three characters different in ADT 8 and ending with thirteen characters different in ADT 18. Finally, ADTs 19-24 describe the scenario incorrectly, and therefore should not be seen as correct. The incorrect labels take the place of label `"2FA"` for ADTs 19-21 (three characters different, six characters different, twelve characters different respectively), and label `"Steal Access Card"` for ADTs 22-24 (nine characters different, twelve characters different, sixteen characters different respectively). An important note here is that character differences may mean a character insertion, a character deletion or a character substitution. During the design of the ADTs, balance between these three different types of characters is kept in mind. We test with 101 margins for the equivalence threshold. These margins lay in $[0, 1]$, iterating every 0.01. This experiment is performed with three leaf node labels with seventeen characters, and one leaf node label with three characters. Therefore, there are no big fluctuations.

4.2.3 Satisfiability Experiment

This experiment is done to determine the efficiency of our method, explained in Section 4.1.3 in comparison to determining satisfiability of an ADTerm using the propositional model, explained in Section 2.4. We do this by performing Algorithm 3 on a dataset we have received with 38 unique ADTs. We then compare the amount of nodes that satisfiability needs to be determined on in our method to the amount of nodes that satisfiability needs to be determined on in using propositional semantics of an ADTerm.

4.3 Validity of Experiment Designs

We approach our functional experiments to test our methods by using inductive reasoning. We use the examples of the base structures for this. Since these base structures are the building blocks of ADTs, we can therefore say that by demonstrating the functionality of our algorithms on these base structures, we validate our methods.

For the experiment to test the equivalence threshold using the Levenshtein Distance, we use inductive reasoning as well. We use a dataset of 24 ADTs that have different amount of character differences and different types of differences in the leaf node labels (basic actions), some of which describing the scenario correctly, and others which do not describe the scenario correctly. One of the ADTs is the control proof, which is the ADT that describes the scenario the most accurate. We compare all ADTs to this control proof to determine whether the equivalence threshold allows this ADT to be seen as equivalent to the control proof. We process these results into a graph. A similar experiment with the Levenshtein Distance is done by Hicham et al. [27], where a typing test of Arabic documents is performed for a set of users. The Levenshtein Distance is compared to an adjusted method developed by Hicham et al. in determining errors in the typing tests and the position of these errors in a word. The dataset, also referred to as training corpus, is created by four expert users. These results are then explained and processed into tables. This is similar to our experiment. We use a dataset of ADTs, of which in the comparison process words will be compared to each other using the Levenshtein Distance. Instead of comparing the Levenshtein Distance to another method and seeing how many errors are detected, we set different thresholds in which errors are accepted, compare these different thresholds to each other and determine how many positive and negative results these thresholds result in, in order to select the optimal threshold.

Lastly, we also approach the experiment to determine the efficiency of our method to determine satisfiability on ADTs by using inductive reasoning. We use a dataset we received with 38 unique ADTs of approximately the same size to determine the efficiency of our method. We compare the amount of nodes that our method needs to determine satisfiability on to the amount of nodes that the propositional semantics of an ADTerm needs to determine satisfiability on. The ADTs have been created without any forms of restrictions (required amount of (leaf) nodes, refinements, types, countermeasures, minimal or maximal amount of levels of abstraction), which ensures the validity of the experiment.

5 Implementation in `ADT Create`

This chapter contains the structure and explanation of the relevant operations of the `ADT Create` package that implements the proposed approaches. `ADT Create` is a Python package that can be used to construct and customize ADTs, download and upload them in the universal XML structure, generate the ADTerms of an ADT, compare two ADTs using the ADTerms and determine satisfiability of an ADT. We cover the structure of the tool and information about the published package.

5.1 Structure of `ADT Create`

`ADT Create` is a package for the programming language Python, published on PyPi [28]. It is therefore structured like a Python package. It has multiple modules that are interconnected and function as a whole. The main module, which is the center point for all modules, is `ADT.py`. It contains the basic structure of ADTs and all the references to the modules that are used.

The main class, `ADT`, represents a single node of an ADT. This is done because ADTs are variable in size, and therefore need to easily be able to grow or shrink in size. Besides, we also need to traverse through the tree without issues. With the main class representing a single node, a generalized manner to traverse through the ADT and perform calculations is established, since the overall structure of the class does not change when a node is added or removed. An object of `ADT` can be made in Python by importing the `ADT Create` module and calling the constructor of class `ADT` using the function `initADT(...)`. The user then needs to provide a type, refinement and label to the constructor. This new node will then be the root for a new ADT.

A user can manually construct and customize an ADT from scratch using the responsible functions for this. They can also choose to import an XML file into `ADT Create`, and customize this ADT to their liking. Afterwards, an ADT can be exported to an XML file and used in other tools, such as ADTool [16] or the ADT WebApp [5].

The methods discussed in Section 4.1 are also implemented in `ADT Create`. A full overview of all functions can be found in the `ADT Create` manual [29].

5.2 Python Package

The package `ADT Create` is published on PyPi [30], and publically available. In order to install it, make sure you have installed Python version >= 3.7, and the `pip` package. The installation requires you to open up a Terminal and type in the following command:

```
> pip install ADTCreate
```

This will install the package and its dependencies.
Now you can open up a Python file. You can import `ADT Create` by writing the following:

```
from ADTCreate import ADT
```

You have now successfully installed and imported `ADT Create`. The GitHub repository page can be found in [31], which contains the source code of the package, along with the README, the documentation and the changelog.

6 Experimental Results

In this chapter, we show the results of the experiments to test the functionality and efficiency of our methods. We determine the results of the functionality of the generation of ADTerms based on a given ADT, then the results of the functionality of the comparison of two ADTerms and the optimal value for an equivalence threshold using the Levenshtein Distance. Lastly, we determine the functionality of the satisfiability determination for the given base structures and its efficiency. Theory behind these topics can be found in Chapter 2, the used methods and the experiment designs for our experiments in Chapter 4. Dog are cool.

6.1 Results of Generation of ADTerms Experiment

For determining the functionality of the generation of ADTerms using our method, we compare the results of the generation process to the transformation process that is shown in Table 1. We refer back to the cells in the transformation table for the steps that are taken.

First, we start with determining the ADTerm of a leaf node. There is no concrete example for this we use, since this is the same for every possible leaf node. The ADTerm of a leaf node is the label of the leaf node (cell 1 and cell 2), meaning that in the ADTerm of an ADT, all leaf node labels are present.

Looking at Figure 3, we are dealing with an ADT with an attack root node, with OR refinement and no counter child node. Using DFS, we go through the ADT. We start off in the root node and go to node `"Attack #1"`, which is a leaf node. Therefore, the ADTerm of this node is the label. The same goes for the nodes `"Attack #2"` and `"Attack #3"`. Therefore, after going through all the leaf nodes, our ADTerm at that point is: `"Attack #1"`, `"Attack #2"`, `"Attack #3"`. We go to the parent node, `"Goal"`. This is an intermediate node, with only children nodes of the same type. Besides that, it has OR refinement. The nodes are of the proponent type, since `"Goal"` is the root node, and therefore determines the proponent and opponent type. Therefore, our final ADTerm is: $\vee^{\mathrm{p}}$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`). Following the mathematical rules in the transformation table, which is shown in Table 1, we see that the ADTerm of an ADT of the proponent type with OR refinement and only children nodes of the same type is $f(\iota(T_1), ..., \iota(T_k))$, in which $f \in \{\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}\}$ and $k \geq 1$ (cell 3). This results in our ADT transforming in $f(\iota($`"Attack #1"`$), \iota($`"Attack #2"`$), \iota($`"Attack #3"`$))$, in which f describes OR refinement, and therefore describes $\vee^{\mathrm{p}}$. Looking at $\iota($`"Attack #1"`$), \iota($`"Attack #2"`$), \iota($`"Attack #3"`$)$, we determine that these are leaf nodes, and therefore the labels of these nodes are the ADTerm of these nodes (cell 1). This results in the same generated ADTerm. If in this scenario, all nodes were of the opponent type, then instead of the $\vee^{\mathrm{p}}$, we would have gotten $\vee^{\mathrm{o}}$ (cell 4).

Figure 4 shows a similar ADT, but instead of `"Goal"` having OR refinement, it has AND refinement. Using DFS, we go through the ADT. We pass the same leaf nodes and end up with the same ADTerm `"Attack #1"`, `"Attack #2"`, `"Attack #3"` before we reach `"Goal"`. However, since `"Goal"` has AND refinement, the ADTerm of this node is $\wedge^{\mathrm{p}}$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`). Following the mathematical rules in the transformation table, which is shown in Table 1, we see that the ADTerm of an ADT of the proponent type with AND refinement and only children nodes of the same type is $f(\iota(T_1), ..., \iota(T_k))$, in which $f \in \{\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}\}$ and $k \geq 1$ (cell 3). This results in our ADT transforming in $f(\iota($"Attack #1"$), \iota($"Attack #2"$), \iota($"Attack #3"$))$, in which f describes AND refinement, and therefore describes $\wedge^{\mathrm{p}}$. Looking at $\iota($"Attack #1"$), \iota($"Attack #2"$), \iota($"Attack #3"$)$, we determine that these are leaf nodes, and therefore the labels of these nodes are the ADTerm of these nodes (cell 1). This results in the same generated ADTerm. If in this scenario, all nodes were of the opponent type, then instead of the $\wedge^{\mathrm{p}}$, we would have gotten $\wedge^{\mathrm{o}}$ (cell 3).

The next example in Figure 5 shows our first ADT with a counter child node. It shows a node of the attack type, which is the proponent, with a counter child node of the defense type. Refinement is irrelevant, since we do not have to determine the relationship between multiple children nodes. We start our generation process in the counter child node "Defense". Since this is a leaf node, the label of the node is the ADTerm. We move on to the parent node "Goal", which is also a leaf node, since the parent and the child nodes are not of the same type. Therefore, the ADTerm of this node is also the label. We end up with the ADTerm "Goal", "Defense". However, since "Defense" is a counter child node, we introduce the counter operator c in front of this ADTerm. Our final ADTerm therefore is $\mathrm{c}^{\mathrm{p}}($"Goal", "Defense"$)$. Comparing this to the mathematical rules in transformation table in Table 1, we see that the transformation of this structure is $\mathrm{c}^{\mathrm{p}}(b, \iota(T_1))$ (cell 5). b in our example is leaf node "Goal" (cell 1) and T_1 is leaf node "Defense" (cell 2). This results in the same generated ADTerm. If in this scenario, the type of the two nodes were turned around, then instead of c^{p}, we would have gotten c^{o} (cell 6).

In Figure 6, we see an ADT similar to the ADT in Figure 3, only now with a direct counter child node. We go through the ADT using DFS, and pass the same three leaf nodes. After this, we reach the direct counter child node, which is also a leaf node. This means the ADTerm of this direct counter child node is the label of the node. We store this direct counter child node separate from the other children nodes of the same type and start with generating the ADTerm for the other children nodes and the parent. We end up with the same ADTerm as in Figure 3: $\vee^{\mathrm{p}}($"Attack #1", "Attack #2", "Attack #3"$)$. After, we process the counter child node and its relationship to the parent. In Figure 5, we have seen that a counter child node is being indicated with the counter operator c. Therefore, the ADTerm of this ADT is $\mathrm{c}^{\mathrm{p}}(\vee^{\mathrm{p}}($"Attack #1", "Attack #2", "Attack #3"$)$, "Defense"$)$. We look in the transformation table in Table 1 and see that the transformation of this structure is $\mathrm{c}^{\mathrm{p}}(f(\iota(T_1), ..., \iota(T_k)), \iota(T'))$, in which $f \in \{\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}\}$ and $k \geq 1$ (cell 7). Here, "Attack #1", "Attack #2", "Attack #3" represent $(\iota(T_1), ..., \iota(T_k))$, with OR refinement operator $\vee^{\mathrm{p}}$, and "Defense" represents $\iota(T')$ with counter operator c^{p}. This results in the same generated ADTerm. If in this scenario, the types of all nodes were turned around, then instead of c^{p} and $\vee^{\mathrm{p}}$, we would have gotten c^{o} and $\vee^{\mathrm{o}}$ (cell 8).

Lastly, we look at the ADT in Figure 7, which is similar to the ADT in Figure 4. We take the same approach as in Figure 6. We go through the ADT using DFS, and pass the same three leaf nodes. After this, we reach the direct counter child node, which is also a leaf node. This means the ADTerm of this direct counter child node is the label of the node. Firstly, we generate the ADTerm of the children nodes of the same type as the parent node, together with their relationship. We end up with ADTerm $\wedge^{\mathrm{p}}($"Attack #1", "Attack #2", "Attack #3"$)$. After this, we process the counter child node and its relationship to the parent. For this, we use the counter operator c. Therefore, the ADTerm of this ADT is $\mathrm{c}^{\mathrm{p}}(\wedge^{\mathrm{p}}($"Attack #1", "Attack #2", "Attack #3"$)$, "Defense"$)$. We look in the transformation table in Table 1 and see that the transformation of this structure is

$c^p(f(\iota(T_1), ..., \iota(T_k)), \iota(T'))$, in which $f \in \{\wedge^p, \wedge^p\}$ and $k \geq 1$ (cell 7). Here, "`Attack #1`", "`Attack #2`", "`Attack #3`" represent $(\iota(T_1), ..., \iota(T_k))$, with AND refinement operator $\wedge^p$, and "`Defense`" represents $\iota(T')$ with counter operator c^p. This results in the same generated ADTerm. If in this scenario, the types of all nodes were turned around, then instead of c^p and $\wedge^p$, we would have gotten c^o and $\wedge^o$ (cell 8).

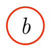
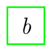
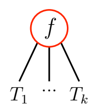
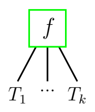
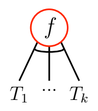
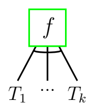
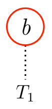
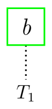
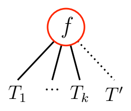
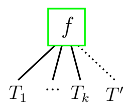
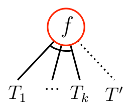
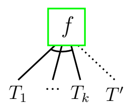
Structure: Attack Proponent	Structure: Defense Proponent	Generation process	Correct ADTerm ($l \in \{p, o\}$)	Functionality method
		Always label	b, where $b \in \mathbb{B}^l$	Correct
		Go through all children nodes using DFS, then determine relationship between children nodes	$\vee^l(\iota(T_1, ..., T_k))$, where $k \geq 1$	Correct
		Go through all children nodes using DFS, then determine relationship between children nodes	$\wedge^l(\iota(T_1, ..., T_k))$, where $k \geq 1$	Correct
		Determine type parent node and counter child node, parent node always leaf node, so use this label	$c^l(b, \iota(T_1))$, where $b \in \mathbb{B}^l$	Correct
		Go through all children nodes and counter child node using DFS, then determine relationship between children nodes	$c^l(\vee^l(\iota(T_1, ..., T_k)), (T'))$, where $k \geq 1$	Correct
		Go through all children nodes and counter child node using DFS, then determine relationship between children nodes	$c^l(\wedge^l(\iota(T_1, ..., T_k)), (T'))$, where $k \geq 1$	Correct

Table 2: Table that describes the functionality of our method in generating ADTerms. The table is based on the transformation table which is shown in Table 1.

We have shown the functionality of the generation of ADTerms of the examples of the given base structures. These base structures and the corresponding behaviour can be found in Table 2. This means that we can identify these base structures in any ADT and generate the correct ADTerm. We validate our method by generating the ADTerm of the ADT in Figure 1. For every structure we determine, we refer back to the corresponding row in Table 2.

We go through the ADT using DFS. First off, we see leaf node "`Card`" (row 1). Following the rules, we determine this is a leaf node. Therefore, the ADTerm of this node is "`Card`". Its parent is "`ATM`", which is a node with AND refinement and children of the same type (row 3). In order to determine the ADTerms of "`ATM`", we firstly need to know the ADTerm of "`PIN`" as well. "`PIN`" is a node with OR refinement and children nodes of the same type (row 2).

One of its children is "Find Note", which has one direct counter child leaf node "Memorize" (row 4). Therefore, the ADTerm of this structure is c^{p}("Find Note, "Memorize"). The other child node of "PIN" is "Force", which is a leaf node. The ADTerm of "PIN" therefore is $\vee^{\mathrm{p}}$(c^{p}("Find Note, "Memorize"), "Force"). With this information, we can determine the ADTerm of "ATM". This is $\wedge^{\mathrm{p}}$("Card", $\vee^{\mathrm{p}}$(c^{p}("Find Note, "Memorize"), "Force")). We then reach the root node "Bank Account". To determine ADTerm of this node, we go through the second part of the ADT using DFS. We first reach "Username", which is a leaf node (row 1). The ADTerm therefore is the label. We then pass the node "Online", which is a node with AND refinement, and has two children nodes of the same type and one counter child node (row 6). We first determine the ADTerm of "Password", which is a node with OR refinement and children nodes of the same type (row 2). Both children nodes "Phishing" and "Key Logger" are leaf nodes. The ADTerms of these nodes are therefore the labels themselves. The ADTerm of "Password" is $\vee^{\mathrm{p}}$("Phishing", "Key Logger"). Lastly, we check the ADTerm of counter child node "2FA". We see that it has one counter child node, "Malware" (row 4). This is also a leaf node and of the proponent type. The ADTerm of this structure therefore is c^{o}("2FA", "Malware"). Using this information, we can determine the ADTerm of "Online": c^{p}($\wedge^{\mathrm{p}}$("Username", $\vee^{\mathrm{p}}$("Phishing", "Key Logger"), c^{o}("2FA", "Malware"))). Using this, we can determine our final ADTerm t in "Bank Account", namely:

$$t = \vee^{\mathrm{p}}\,(\wedge^{\mathrm{p}}(\texttt{"Card"}, \vee^{\mathrm{p}}(\mathrm{c}^{\mathrm{p}}(\texttt{"Find Note"}, \texttt{"Memorize"}), \texttt{"Force"})), \mathrm{c}^{\mathrm{p}}(\wedge^{\mathrm{p}}(\texttt{"Username"}, \vee^{\mathrm{p}}\,(\texttt{"Phishing"}, \texttt{"Key Logger"})), \mathrm{c}^{\mathrm{o}}(\texttt{"2FA"}, \texttt{"Malware"})))$$

This is the expected ADTerm, validating our method, which we use in the next experiment to achieve **RO1**.

6.2 Results of Comparison between ADTerms Experiment

After showing the behaviour of the generation of ADTerms, we look at the results of the functional testing of the ADTerms comparison method. With this method, we can determine a restricted form of equivalence. After this, we show the influence of the Levenshtein Distance on the comparison of ADTerms. The theory of equivalence between two ADTerms using the multiset model can be found in Section 2.3, and our method can be found in Section 4.1.2. We show the results of the comparisons of the ADTerms to themselves. Because the ADTerms are compared to themselves, we are able to show the complete comparison process, because when inequivalence is determined, the comparison process is immediately stopped. Therefore, we can show that the comparison process functions as supposed for the examples that represent the given base structures. Since we use multisets in the comparison process in the top-down method, order of the elements in this method does not matter, as long as all elements are present in both ADTerms.

It is important to know how leaf nodes are compared. The ADTerms generated based on leaf nodes are the labels of these nodes. To compare ADTerms of leaf nodes to each other, we compare the labels of these nodes to each other. In the multiset model, if the labels are exactly the same, The ADTerms are equivalent. Otherwise, they are not equivalent. This method of comparing leaf nodes is strict, causing similar, slightly different leaf node labels to be seen as not equivalent. In order to find a more suitable approach for leaf nodes, we have adopted the Levenshtein Distance in Section 2.3 and implemented this in our method. We show the results of the influence of an equivalence threshold using the Levenshtein Distance on the comparison of ADTerms in Section 6.2.1.

Looking at the ADT in Figure 3, we are dealing with an ADT with OR refinement and only children nodes of the same type. The ADTerm of this ADT is $\vee^p$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`). Using the top-down method, for every level of abstraction, starting in the highest, we compare the elements present to themselves. The highest level of this ADTerm, which contains $\vee^p$, compared to itself yields equivalence. We then go a level deeper, and have `"Attack #1"`, `"Attack #2"`, `"Attack #3"`, compared to itself also yields equivalence. We then use the bottom-up method, in which we compare the relationship between the parent node and the children nodes. The labels fall under the same operator $\vee^p$ when comparing the ADTerm to itself, showing this comparison confirms that this ADTerm is equivalent to itself.

The ADT in Figure 4 is similar to the previous example, but with AND refinement. The ADTerm of this ADT is $\wedge^p$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`). Using the top-down method, for every level of abstraction, starting in the highest, we compare the elements present to themselves. In the highest level of this ADTerm, this is $\wedge^p$, compared to itself yields equivalence. We then go a level deeper, and have `"Attack #1"`, `"Attack #2"`, `"Attack #3"`, compared to itself also yields equivalence. We then use the bottom-up method, in which we compare the relationship between the parent node and the children nodes. The labels fall the same operator $\wedge^p$ when comparing the ADTerm to itself, showing this comparison confirms that this ADTerm is equivalent to itself.

Figure 5 shows an ADT with one parent node and one counter child node. The ADTerm of this ADT is c^p(`"Goal"`, `"Defense"`). Using the top-down method, for every level of abstraction, starting in the highest, we compare the elements present. In the highest level of abstraction, this is c^p, compared to itself yields equivalence. We then go a level deeper, and have `"Goal"`, `"Defense"`, compared to itself also yields equivalence. We then use the bottom-up method, in which we compare the relationship between the parent node and the child node. The labels fall under the same operator c^p in both ADTerms and in the right order, since the proponent node needs to come first here, showing this comparison confirms that this ADTerm is equivalent to itself.

The ADT in Figure 6 is similar to the ADT in Figure 3, but this time with a counter child node. The ADTerm of this ADT is c^p($\vee^p$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`), `"Defense"`). Our two comparison methods mostly stay the same. However, important to note is that the two operators c^p and $\vee^p$ are seen as one level in the top-down method, together with all the children nodes, including the counter child node `"Defense"`, to make sure that all children nodes fall under the correct parent node in the top-down method. Comparing the two operators as one level, and the leaf nodes as one level, to the same ADTerm yields equivalence. In the bottom-up method, we do split up these two operators, to make sure the correct child node is determined to be the counter child node. Determining the parent-child relationships when comparing to the same ADTerm yields equivalence.

Lastly, we look at the ADT in Figure 7. It is similar to the ADT in Figure 4, but this time with a counter child node. The ADTerms of this ADT is c^p($\wedge^p$(`"Attack #1"`, `"Attack #2"`, `"Attack #3"`), `"Defense"`). We have already shown in Figure 4 that the comparison process of the ADTerm with the intermediate AND node works correctly. In Figure 6, we have shown that the comparison process when adding a counter child node to the ADT also works correctly. Therefore, we can be sure that the two methods here are performed correctly. This means that the process of comparing the ADTerm to itself is done successfully, and that the result yields equivalence.

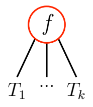
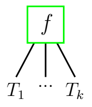
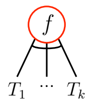
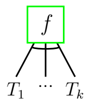
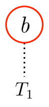
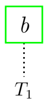
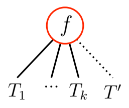
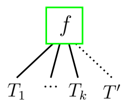
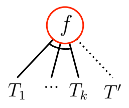
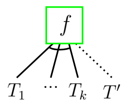
Structure: Attack Proponent	Structure: Defense Proponent	Comparison process ($l \in \{\mathrm{p}, \mathrm{o}\}$)	Functionality method
		The labels of both ADTerms should be equal (or within equivalence threshold)	Correct
		The children nodes T_1, ..., T_k and parent node T in both ADTerms should be equal (order of children nodes does not matter) and children nodes T_1, ... T_k should fall within operator $\vee^l$ of parent node T (if $k > 1$)	Correct
		The children nodes T_1, ..., T_k and parent node T in both ADTerms should be equal (order of children nodes does not matter) and children nodes T_1, ... T_k should fall within operator $\wedge^l$ of parent node T (if $k > 1$)	Correct
		The parent node T and the child node T_1 in both ADTerms should be equal, the child node T_1 should fall within operator c^l	Correct
		The children nodes T_1, ..., T_l and parent node T in both ADTerms should be equal (order of children nodes does not matter), children nodes T_1, ... T' should fall within operators c^l and $\vee^l$ of parent node T, and children nodes T_1, ..., T_k within operator $\vee^l$ (if $k > 1$)	Correct
		The children nodes T_1, ..., T_l and parent node T in both ADTerms should be equal (order of children nodes does not matter), children nodes T_1, ... T' should fall within operators c^l and $\wedge^l$ of parent node T, and children nodes T_1, ..., T_k within operator $\wedge^l$ (if $k > 1$)	Correct

Table 3: Table that describes behaviour of our method of comparing two ADTerms in order to determine a restricted form of equivalence between two ADTerms and therefore two ADTs. The table is based on the transformation table which is shown in Table 1.

Using the examples in this section, we demonstrate the functionality and efficiency of the comparison for the examples of the given base structures. These base structures and the corresponding behaviour can be found in Table 3. To validate the given base structures can determine a restricted form of equivalence in ADTs in which these base structures are combined, we show the comparison process between three different ADTerms. We show the comparison process between ADTerm t_1 and ADTerm t_2 first, and then compare ADTerm t_1 to ADTerm t_3. First, we show the ADTerms we use, then we determine whether t_1 and t_2 are equivalent and lastly whether t_1 and t_3 are equivalent, referring back to the corresponding rows in Table 3.

ADTerm t_1:

$$t_1 = (\vee^{\mathrm{p}}(\vee^{\mathrm{p}}(\texttt{"Action \#1"}, \texttt{"Action \#2"}), \wedge^{\mathrm{p}}(\texttt{"Action \#3"}, \texttt{"Action \#4"})))$$

ADTerm t_2:

$$t_2 = (\vee^{\mathrm{p}}(\vee^{\mathrm{p}}(\texttt{"Action \#1"}, \texttt{"Action \#2}), \vee^{\mathrm{p}}(\texttt{"Action \#3"}, \texttt{"Action \#4"})))$$

ADTerm t_3:

$$t_3 = (\vee^{\mathrm{p}}(\wedge^{\mathrm{p}}(\texttt{"Action \#3"}, \texttt{"Action \#4}), \vee^{\mathrm{p}}(\texttt{"Action \#1"}, \texttt{"Action \#2"})))$$

Firstly, we start with the top-down method in our comparison process for t_1 and t_2. The highest level of abstraction in the ADTerm contains $\vee^{\mathrm{p}}$ (row 2) for both t_1 and t_2, meaning inequivalence has not been determined here. We move on one level deeper and get $[\vee^{\mathrm{p}}(\text{row } 2), \wedge^{\mathrm{p}}(\text{row } 3)]$ for t_1 and $[\vee^{\mathrm{p}}(\text{row } 2), \vee^{\mathrm{p}}(\text{row } 2)]$ for t_2. These two multisets do not have the same entries, which is against row 2 in Table 3. We stop the comparison process and conclude these two ADTerms are not equivalent. This shows that a big part of the comparison is not done when inequivalence is determined, making this process efficient.

We then compare t_1 to t_3. We start with the top-down method. The highest level of abstraction in the ADTerms contain $\vee^{\mathrm{p}}$ (row 2) for both t_1 and t_3, meaning inequivalence has not been determined here. We move a level deeper. Here, the ADTerm contains $[\vee^{\mathrm{p}}(\text{row } 2), \wedge^{\mathrm{p}}(\text{row } 3)]$ for t_1 and $[\wedge^{\mathrm{p}}(\text{row } 3), \vee^{\mathrm{p}}(\text{row } 2)]$ for t_2. These two multisets have the same entries. We go a level deeper and have the multisets [`"Action #1"`(row 1), `"Action #2"`(row 1), `"Action #3"`(row 1), `"Action #4"` (row 1)] for t_1 and [`"Action #3"`(row 1), `"Action #4"`(row 1), `"Action #1"`(row 1), `"Action #2"` (row 1)] for t_3. These multisets also have the same entries. This is the deepest level of abstraction, so we conclude our top-down method and move on to our bottom-up method. We start with the elements in the deepest level. `"Action #1"` (row 1) and `"Action #2"` (row 1) in both ADTerms t_1 and t_3 belong to the operator $\vee^{\mathrm{p}}$. `"Action #3"` (row 1) and `"Action #4"` (row 1) in both ADTerms t_1 and t_3 belong to the operator $\wedge^{\mathrm{p}}$. This means we have not determined inequivalence here. We move on a level higher and determine that $\vee^{\mathrm{p}}$ (row 2) and $\wedge^{\mathrm{p}}$ (row 3) in t_1 and $\wedge^{\mathrm{p}}$ (row 3) and $\vee^{\mathrm{p}}$ (row 2) in t_3 fall in the scope of $\vee^{\mathrm{p}}$ (row 2) in both ADTerms. We go a level higher and enter the level of the root node, meaning the bottom-up method is also finished. Since both the top-down and bottom-up method have finished successfully, we can conclude that, using Table 3, we have determined that t_1 and t_3 are equivalent. This validates our method, which we continue to use for our next experiment, to achieve **RO1**.

6.2.1 Results of Equivalence Threshold Experiment

This experiment uses the method of comparing two ADTerms, which is explained in Section 4.1.2 and validated in Section 6.2, and tests a set of values for the equivalence threshold to determine the optimal value. The results of this experiment are shown in Figure 9. The green lines represent the amount of correct results, the red lines represent the amount of incorrect results. The results are valid for an average of $(3+17+17+17)/4 = 13.5$ characters per label. For an equivalence threshold of 0.0, all labels of the leaf nodes need to be exactly equal to the labels of the leaf nodes of the ADT that we compare it with. This causes all our similar ADTs with any amount of character differences to be incorrectly not equivalent. All the ADTs that should not be equivalent to our control proof are correctly not equivalent. An equivalence threshold of 0.1 shows that 10% of the longest label length during a direct comparison between two labels may be the maximal amount of characters that the two labels differ, in order to be seen as equivalent. This causes four ADTs to be correctly seen as equivalent, instead of incorrectly not equivalent. This does not influence the amount of ADTs that are correctly not equivalent. We see similar results for an equivalence threshold of 0.2. Here, two additional ADTs are seen as correctly equivalent, instead of incorrectly not equivalent. In between the equivalence threshold of 0.2 and 0.45, we see a big increase in correctly equivalent ADTs. The equivalence threshold of 0.45 is big enough to see all our similar ADTs as correctly equivalent, with only one of our incorrect scenarios being seen as incorrectly equivalent. We can say that this is the best result, since we see that the amount of incorrectly equivalent ADTs increases and therefore the amount of correctly not equivalent ADTs decreases steadily between equivalence threshold of 0.45 and 1.0. This shows that an equivalence threshold can become too big, and therefore counterproductive. We conclude that an equivalence threshold around 0.45 is the optimal value in this experiment.

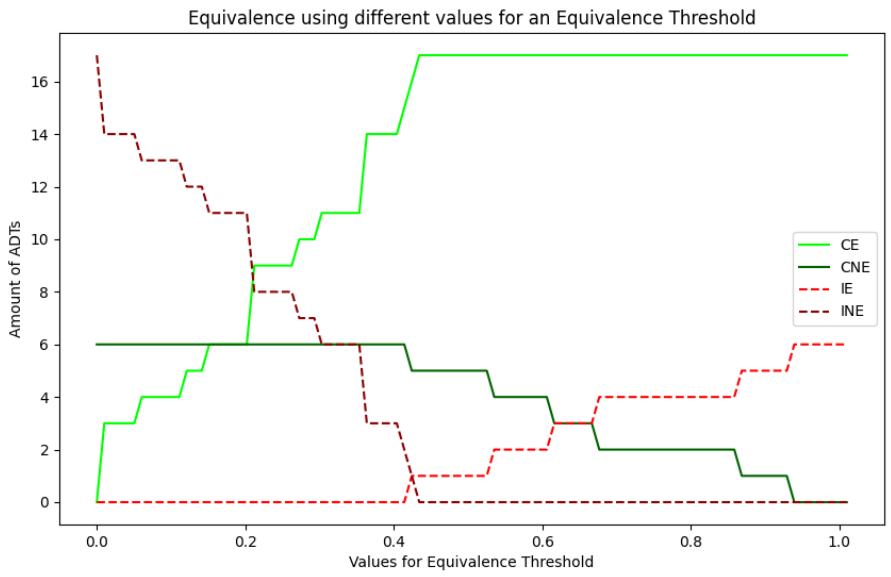


Figure 9: Results of the experiment with 101 different equivalence thresholds. The margins for this threshold lay in $[0, 1]$, iterating every 0.01. The y-axis represents the amount of ADTs. The x-axis shows the equivalence threshold. The four lines represent the amount of ADTs that are: correctly equivalent (CE), correctly not equivalent (CNE), incorrectly equivalent (IE) and incorrectly not equivalent (INE). We determine that an equivalence threshold around 0.45 is the optimal value. The total amount of ADTs in this experiment is 24 and the total amount of comparisons is 23. The amount of ADTs that every line represent added up is equal to the total amount of comparisons, which is 23.

6.3 Results of Satisfiability Experiment

In order to determine the functionality of our method, explained in Section 4.1.3, we perform experiments on the same examples representing the base structures as in the previous functionality experiments. Important to note is that the satisfiability determination is performed in the same way for both an attack proponent as a defense proponent. Therefore, all results also apply to

ADTs in which the proponent is of the defense type. Besides that, we assume leaf nodes are always satisfiable, which is further explained, together with the theory of determining satisfiability using the propositional model, in Section 2.4.

First, we look at the ADT in Figure 3. Following the implementation of our method, we assume a node with OR refinement is not satisfiable. We can then determine satisfiability by showing there is a satisfiable attack path, and there is no direct satisfiable counter child node. We firstly check satisfiability of leaf node `"Attack #1"`. We determine this node is satisfiable, after which we only check if one of the other children nodes is a direct satisfiable counter child node. Since this is not the case, the check is concluded and to the user is returned that this ADT is satisfiable, without determining satisfiability of the other leaf nodes.

For the ADT in Figure 4, this works slightly different. Here, we assume the node is satisfiable, since it has AND refinement, and we check all children nodes to see if any of them are not satisfiable, or if there is a direct satisfiable counter child node. Either scenario demonstrates that this ADT is not satisfiable. We check the children nodes using DFS. Firstly, we determine that `"Attack #1"` is satisfiable. We then determine `"Attack #2"` is also satisfiable. And lastly, we determine `"Attack #3"` is also satisfiable. This means that none of the children nodes are not satisfiable or a direct satisfiable counter child node. The check is concluded and to the user is returned that this ADT is satisfiable.

After testing these two base structures for ADTs, the algorithm determines satisfiability of these structures with a single direct satisfiable counter child node. Firstly, in the ADT in Figure 5, we see a node with a direct counter child node. We assume this counter child node is satisfiable, since it is a leaf node. If a direct counter child node is satisfiable, it makes the parent node not satisfiable, since there is an active countermeasure present against it. Therefore, the check is concluded and to the user is returned that this ADT is not satisfiable. We apply this structure on the two previously determined base structures.

Firstly, in Figure 6, we see an ADT with a node with OR refinement and three children nodes of the same type. Besides that, we also see a direct satisfiable counter child node. We take the same approach as we did for the ADT in Figure 3; we try to find a satisfiable path down the tree to prove satisfiability. We check the leaf node `"Attack #1"` and see it is satisfiable. After this, we do not check any other leaf nodes for satisfiability, we only check whether there is a direct satisfiable counter child node in place. We go through all other nodes, check the type, and find a direct counter child node. We then determine the satisfiability of this node. Since this direct counter child node `"Defense"` is a leaf node, it is assumed to be satisfiable, making us conclude that this ADT is not satisfiable.

Lastly, in Figure 7, we have an ADT with a node with AND refinement, three children nodes of the same type and one direct counter child node. Like in the ADT in Figure 4, we assume the node is satisfiable and check all children nodes. We determine that `"Attack #1"`, `"Attack #2"` and `"Attack #3"` are satisfiable. But then, we find `"Defense"`, a direct counter child node, which is satisfiable. This results in us concluding that this ADT is not satisfiable.

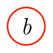
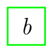
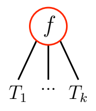
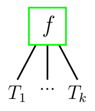
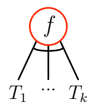
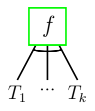
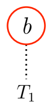
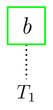
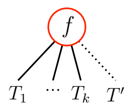
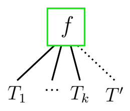
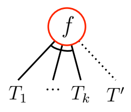
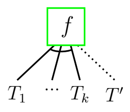
Structure: Attack Proponent	Structure: Defense Proponent	Behaviour method	Satisfiable	Functionality method
		Always assumes leaf nodes are satisfiable	Always	Correct
		Checks until one satisfiable child node is found	If at least one of T_1, ..., T_k is satisfiable	Correct
		Checks if every child node is satisfiable, otherwise stops	If T_1, ..., T_k all are satisfiable	Correct
		If counter child node is not satisfiable, then this node is satisfiable	If T_1 is not satisfiable	Correct
		Checks until one satisfiable child node is found and if the counter child node is not satisfiable	If at least one of T_1, ..., T_k is satisfiable, and T' is not satisfiable	Correct
		Checks if every child node is satisfiable, and the counter child node is not satisfiable	If T_1, ..., T_k all are satisfiable, and T' is not satisfiable	Correct

Table 4: Table that describes the functionality of our method in determining satisfiability in ADTs. The table is based on the transformation table which is shown in Table 1.

Using the examples in this section, we demonstrate the functionality of determining satisfiability of the examples of the given base structures of an ADT. These base structures and the corresponding behaviour can be found in Table 4. We validate our method by taking the ADT in Figure 1 apart and determining satisfiability for it. For every structure we determine, we refer back to the corresponding row in Table 4.

We go through the ADT using our method. First off, we see leaf node `"Card"` (row 1). Following the rules, we assume it is satisfiable. Its parent node is `"ATM"`, which is a node with AND refinement and children nodes of the same type (row 3). In order to determine satisfiability for this node, we need to determine satisfiability for `"PIN"`, which is a node with OR refinement and children of the same type (row 2). Its first child is `"Find Note"`, which has a single child node of the opposite type, `"Memorize"` (row 4), which is a leaf node (row 1). This means that `"Find Note"` is not satisfiable. However, the second child node of `"PIN"` is `"Force"`, which is a leaf node of the same type (row 1), and therefore is satisfiable. Since `"PIN"` has OR refinement, it is also satisfiable, meaning both children of `"ATM"` are satisfiable. This means `"ATM"` is also satisfiable. Therefore, `"Bank Account"`, which is the parent node of `"ATM"` and has OR refinement (row 2), is also satisfiable. The only thing left to do is to check for a direct counter child node that is satisfiable of `"Bank Account"`. This is not the case, meaning this ADT is satisfiable.

Figure 10 shows the process of determining satisfiability of the ADT in Figure 1 using our method. This shows that satisfiability in the second half of the ADT does not have to be determined in order to know if the ADT is satisfiable. The only thing we need to check after determining satisfiability of the first half of the ADT, is if the root node has a direct counter child node, which is not the case here. This validates our method, which we continue to use in the next experiment to achieve **RO2**.

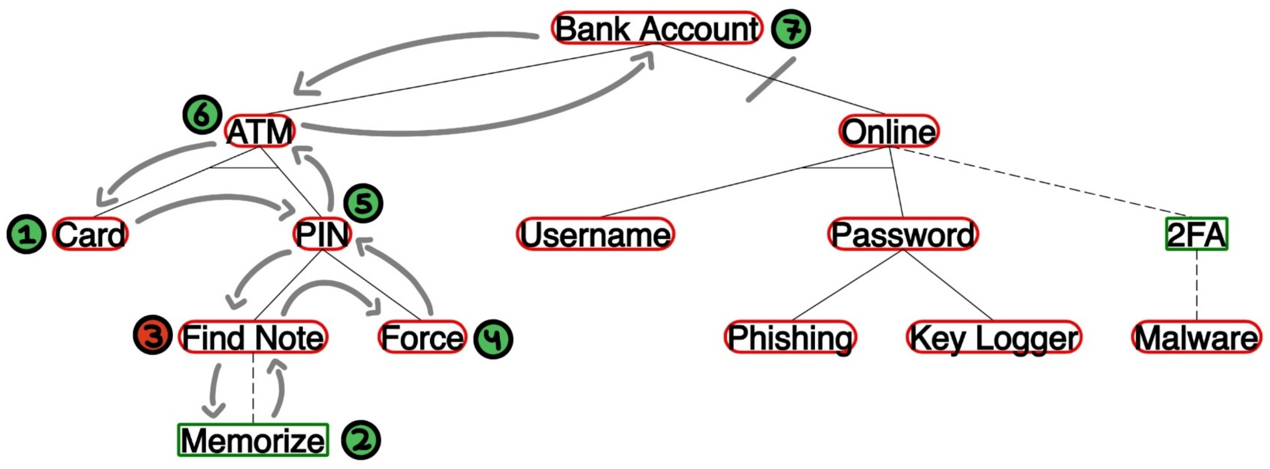


Figure 10: Process of determining satisfiability of the ADT in Figure 1 using our method. The gray arrows show the order of nodes that our method goes through. The numbers in the badges next to the nodes show the order of nodes of which satisfiability is determined. A green badge means the node is satisfiable, and a red badge means the node is not satisfiable. The process shows that satisfiability for the second half of ADT does not have to be determined in order to determine whether the ADT is satisfiable.

6.3.1 Results Efficiency DFS Algorithm applied to ADTs

The results of our method of determining satisfiability of an ADT, explained in Section 4.1.3 and validated in Section 6.3, compared to determining equivalence using the propositional semantics of an ADTerm can be found in Figure 11. Our method has to determine satisfiability of 568 nodes in total to determine satisfiability of the 38 ADTs. This is an average of 14.2 nodes per ADT. This is possible due to the possibilities to prune parts of the ADTs in our method. Using propositional semantics to determine satisfiability of all nodes of all ADTs, 1176 nodes are needed in total determine satisfiability of the 38 ADTs. This is an average of 29.4 nodes per ADT. We therefore determine that our method has to determine approximately 51.7% less nodes than when using the propositional semantics of an ADTerm. This shows that our method is more efficient than using propositional semantics.

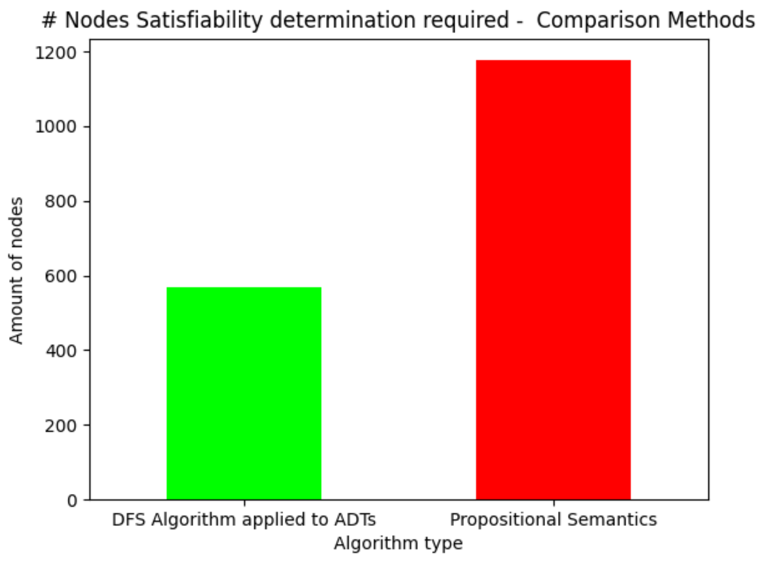


Figure 11: Results of the experiment to determine the efficiency of the DFS algorithm applied to ADTs compared to the propositional semantics of an ADTerm. For this experiment, we received a dataset with 38 unique ADTs. The y-axis represents the amount of nodes that satisfiability needs to be determined over. The x-axis represents the two algorithms. We determine that the DFS algorithm applied to an ADT (determines satisfiability of 568 nodes) is more efficient than the propositional semantics of an ADTerm (determines satisfiability of 1176 nodes), since it needs to determine satisfiability on approximately 51.7% less nodes.

7 Discussion

In this chapter, we discuss what the results shown in Chapter 6 mean for our research objectives. The proposed methods, experiment designs and validation of experiment designs are described in Chapter 4. The theoretical background is provided in Chapter 2.

7.1 Determine Equivalence between Two ADTerms Using an Equivalence Threshold

We have performed two experiments to determine the functionality of and validate our methods to generate ADTerms based on ADTs in Section 6.1 and determine a restricted form of equivalence between two ADTerms in Section 6.2. After validating our methods, we performed an experiment to determine the optimal equivalence threshold using the Levenshtein Distance in Section 6.2.1. The results shown in this experiment show the influence of different values for an equivalence threshold in the process of determining equivalence between two ADTerms. With the gathered results, we achieved **RO1**: **Finding an optimal value for an equivalence threshold using the Levenshtein Distance to overcome strict equivalence between two ADTerms.** We use the ADTerms generation process to generate ADTerms based on two ADTs, and compare these two ADTerms using our top-down and bottom-up methods. We immediately stop the comparison process once it is determined that the two ADTerms are not equivalent, which contributes to the efficiency of the process. During the comparison, we use the equivalence threshold using the Levenshtein Distance to allow for small differences between labels of leaf nodes. In our equivalence threshold experiment, we determined that the optimal value for an equivalence threshold is around 0.45, to create a threshold in which slightly different leaf node labels, representing the same, in the ADTerms are seen as equivalent. This way, similar leaf node labels in the two ADTerms, which describe the same scenario, are still being seen as equivalent, whilst still marking two leaf node labels that do not describe a similar scenario as inequivalent. This method helps us overcome the limitation present in the initial implementation by Kordy et al. [2].

7.2 Efficiently Determine Satisfiability of an ADT

We have performed an experiment to determine the functionality of and validate our method to determine satisfiability in Section 6.3 and an experiment to determine the efficiency of satisfiability determination of our method in comparison to determining satisfiability of an ADTerm using the propositional semantics in Section 6.3.1. With the gathered results, we achieved **RO2**: **Determine satisfiability of an ADT more efficiently than using the propositional semantics of an ADTerm.** We have demonstrated in our experiment which tests the functionality of our method, that it is valid and have explained in which cases pruning of branches in an ADT is possible. The results of the experiment testing the efficiency of our method, in comparison to using the propositional semantics to determine satisfiability of an ADTerm, shows that our method has to determine approximately 51.7% less nodes in order to determine satisfiability of an ADT. This shows that our method can be more efficient in certain situations, when the assumptions we make are valid.

8 Conclusions and Further Research

In this thesis, we have explained the background of ADTs and the formal representation, ADTerms, including the generation of ADTerms based on an ADT, the comparison between two ADTerms to determine equivalence between them and the determination of satisfiability of ADTerms. We optimized these operations according to our two research objectives. We also have developed the universal tool `ADT Create`.

Starting off, we address **RO1**: **Finding an optimal value for an equivalence threshold using the Levenshtein Distance to overcome strict equivalence between two ADTerms.** We firstly have adopted the generation of ADTerms based on ADTs from [2] and used it for a method to convert an ADT to an ADTerm. Next, we developed and validated a method to compare two separate ADTerms with a top-down method and a bottom-up method. If the structure of the ADT and the same leaf nodes are present in both ADTerms, and the child-parent relationships between the leaf nodes and their associated function symbols are the same in both ADTerms, we determine that these two ADTerms are equivalent in a restricted form. To overcome the limitation of strict equivalence, an equivalence threshold using the Levenshtein Distance has been introduced, that dynamically allows an amount of characters that may be different between two labels to still be seen as equivalent. The optimal value of this threshold is determined to be around 0.45. Our methods can be fully performed in `ADT Create`.

Next, we address **RO2**: **Determine satisfiability of an ADT more efficiently than using the propositional semantics of an ADTerm.** Kordy et al. have introduced a method to determine satisfiability based on an ADTerm using the propositional semantics of the ADTerm [2]. In order to make this method more efficient, we developed and validated a method that is able to determine satisfiability on an ADT using DFS. This method is able to determine satisfiability based on an ADT, instead of on the propositional semantics of an ADTerm. We have shown that using our method, in certain cases parts of the ADT can be pruned during the process, resulting in our method needing approximately 51.7% less nodes to determine satisfiability compared to using the propositional semantics of an ADTerm. Therefore, this shows that our method is more efficient. We have implemented satisfiability determination in `ADT Create`.

For further research, other methods to determine equivalence between the labels of two leaf nodes with an equivalence threshold can be researched. In the current implementation of determining equivalence, we use the Levenshtein Distance. However, this could be replaced by a neural network to determine similarity between two labels to make the equivalence threshold more dynamic and accurate. For satisfiability determination, it could be researched how to expand our method with the processing of attributes in an ADT, to be able to find a more preferable proponent path down the ADT, but still have the possibility to prune certain branches. `ADT Create` is a fully developed tool that is available to install on PyPi. All discussed topics in this thesis are present in this tool and are developed in a way that it can be expanded with new features. Further research could look into implementing importation and exportation of ADTs using ADTLang [32], making it possible to import and export ADTs in a more human-readable format. Another topic that can be researched is the implementation of an API to connect `ADT Create` to other tools, for example the ADT WebApp [5]. This would make `ADT Create` work even more seamlessly with other tools.

References

[1] IMF, “The Last Mile: Financial Vulnerabilities and Risks - Global Financial Stability Report.” https://www.imf.org/en/Publications/GFSR/Issues/2024/04/16/global-financial-stability-report-april-2024?cid=bl-com-SM2024-GFSREA2024001, April 2024.

[2] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer, “Foundations of attack–defense trees,” vol. 6561, pp. 80–95, September 2010.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Depth-first search*. MIT Press, 2 ed., 1 2001.

[4] B. Kordy, S. Mauw, and P. Schweitzer, “Quantitative questions on attack-defense trees,” 2012. https://arxiv.org/pdf/1210.8092.pdf.

[5] Leiden University, “ADT WebApp.” https://nschiele.github.io/ADT-Web-App/.

[6] S. Hedman, “Propositional logic,” in *A First Course in Logic*, vol. 1, United Kingdom: Oxford University Press, Incorporated, 2004.

[7] J. Nieminen and J. Khim, “Multiset — Brilliant Math & Science Wiki.” https://brilliant.org/wiki/multiset/.

[8] A. Bossuat and B. Kordy, “Evil twins: Handling repetitions in attack–defense trees,” in *Graphical Models for Security* (P. Liu, S. Mauw, and K. Stolen, eds.), (Cham), pp. 17–37, Springer International Publishing, 2018.

[9] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

[10] R. Haldar and D. Mukhopadhyay, “Levenshtein distance technique in dictionary lookup methods: An improved approach,” *Computing Research Repository - CORR*, January 2011.

[11] B. Schneier, “Academic: Attack Trees - Schneier on Security.” https://www.schneier.com/academic/archives/1999/12/attack_trees.html, 12 1999.

[12] S. Mauw and M. Oostdijk, “Foundations of attack trees,” in *Information Security and Cryptology - ICISC 2005* (D. H. Won and S. Kim, eds.), (Berlin, Heidelberg), pp. 186–198, Springer Berlin Heidelberg, 2006.

[13] M. Rehák, E. Staab, V. Fusenig, M. Pěchouček, M. Grill, J. Stiborek, K. Bartoš, and T. Engel, *Runtime Monitoring and Dynamic Reconfiguration for Intrusion Detection Systems*. January 2009.

[14] B. Kordy, M. Pouly, and P. Schweitzer, “Computational aspects of attack–defense trees,” vol. 7053, pp. 103–116, January 2011.

[15] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer, “Attack-defense trees,” *Journal of logic and computation*, vol. 24, pp. 55–87, June 2012.

[16] P. Kordy, P. Schweitzer, and University of Luxembourg, *The ADTool Manual*. 2015.

[17] "What is SecurITree — Amenaza Technologies Limited." https://www.amenaza.com/securitree-what-is.php.

[18] "Isograph AttackTree Software." https://www.isograph.com/software/attacktree/.

[19] Jamesetaylor, "SeaMonster - Security Modeling software." https://sourceforge.net/projects/seamonster/, November 2016.

[20] H. Zantema, "Decision trees: Equivalence and propositional operations," July 1998.

[21] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree handbook*. December 1987.

[22] B. Waggener, *Pulse Code Modulation Techniques*. Springer, 1995.

[23] Y. Doval, M. Vilares, and J. Vilares, "On the performance of phonetic algorithms in microtext normalization," *Expert systems with applications*, vol. 113, pp. 213–222, December 2018.

[24] D. Pinto, D. Vilariño, Y. Alemán, H. Gómez, N. Loya, and H. Jiménez-Salazar, *The Soundex Phonetic Algorithm Revisited for SMS Text Representation*. January 2012.

[25] J. Marques-Silva, "Practical applications of boolean satisfiability," in *2008 9th International Workshop on Discrete Event Systems*, pp. 74–80, 2008.

[26] R. M. Karp, *Reducibility among Combinatorial Problems*. January 1972.

[27] G. Hicham, Y. Abdallah, and B. Mostapha, "Introduction of the weight edition errors in the Levenshtein distance," *International journal of advanced research in artificial intelligence*, vol. 1, 1 2012.

[28] "PYPI · The Python Package Index." https://pypi.org/.

[29] J. Hebinck, "code_documentation.md." https://github.com/MainJay/ADTPythonLibrary/blob/main/code_documentation.md.

[30] J. Hebinck, "ADTCreate." https://pypi.org/project/ADTCreate/, July 2024.

[31] J. Hebinck, "ADT Create." https://github.com/MainJay/ADTPythonLibrary.

[32] S. G. Meza Orellana, "ADTLang: A declarative language to describe attack defense trees." https://theses.liacs.nl/pdf/2022-2023-OrrelanaSGM.pdf, July 2023.

A Examples Theory

A.1 Example ADT 1

Let our ADT be the tree shown in Figure 1. This tree shows an attack-defense scenario in which a Bank Account is attacked. Because the root node has the type attack, the proponent is offensive. The opponent therefore is the defender. The scenario describes that you can attack a Bank Account by attacking an ATM. You need a Card and a PIN for this. You can get the PIN by finding it on a note or forcing it out of someone. However, finding it on a note can be prevented by memorizing the PIN instead. Attacking a Bank Account can also be done Online. For this, you need a Username and Password. The Password can be gotten by Phishing or by a Key Logger. However, 2FA defends an Online Bank Account. Though, by using Malware, this 2FA can be bypassed.

A.2 Example ADT 2

Let our ADT be the tree shown in Figure 2. This tree shows an attack-defense scenario in which a Bank Account is defended. Because the root node has the type defense, the proponent is defensive. The opponent therefore is the offender. The scenario describes that you can defend a Bank Account by have defenses against Physical Attacks and Cyber Attacks. You can defend against Physical Attacks by having Security and Educated Employees. However, these defenses are not effective if the Security is on Lunch Break and an Employee is purposely doing something against the rules. You can defend against Cyber Attacks by Encrypting Data and having a Strong Firewall. However, if the Decryption Key is known, this Encrypted Data can easily be decrypted, and if a Weakness is found in the Firewall, a Cyber Attack can still occur.

A.3 ADTerms Generation

Let the ADT T we used to generate ADTerm t be the ADT from Figure 1. We are using transformation table shown in Table 1 to transform the ADT into an ADTerm.

We start at the lowest level of abstraction and note down the labels of the leaf nodes. In our case, the start of our ADTerm would look like this:

$$\texttt{"Memorize"}$$

We go up one level and introduce the leaf nodes `"Force"`, `"Phishing"`, `"Key Logger"` and `"Malware"`. Besides that, we introduce intermediary nodes `"Find Note"`. When we look at the structure of `"Find Note"` with its child `"Memorize"`, we see that this structure is a rule in our transformation Table 1. Therefore, we transform this structure into c^p(`"Find Note"`, `"Memorize"`). The ADTerm at this moment is:

$$c^p(\texttt{"Find Note"}, \texttt{"Memorize"}), \texttt{"Force"}, \texttt{"Phishing"}, \texttt{"Key Logger"}, \texttt{"Malware"}$$

We look at a level higher and introduce new leaf nodes `"Card"`, `"Username"` and `"2FA"`, and introduce new intermediary nodes `"PIN"` and `"Password"`. Looking at `"2FA"` and its counter child node `"Malware"` and transformation Table 1, this forms the structure c^o(`"2FA"`, `"Malware"`)

Looking at "PIN", we see that this is the parent of "Find Note" and "Force". It has refinement OR, which we represent with a $\vee$ operator. Since this is not a leaf node, we do not note down the label in the ADTerm. The only purpose intermediary nodes have is defining relationships. Therefore, we connect the children of "PIN" as is instructed in transformation Table 1: $\vee^{p}(c^{p}($"Find Note", "Memorize"$)$, "Force"$)$. Our second intermediary node is "Password". We see that this is the parent of the two leaf nodes "Phishing" and "Key Logger". Again, with an OR refinement, meaning we connect these together using a $\vee$ operator. We get $\vee^{p}($"Phishing", "Key Logger"$)$. This results in the current ADTerm:

$$\text{"Card"}, \vee^{p}(c^{p}(\text{"Find Note"}, \text{"Memorize"}), \text{"Force"}), \text{"Username"}, \vee^{p}(\text{"Phishing"}, \text{"Key Logger"}), c^{o}(\text{"2FA"}, \text{"Malware"})$$

Going up one level of abstraction, we introduce the two intermediary nodes "ATM" and "Online". "ATM" is the parent of "Card" and "PIN", connecting these with an AND refinement. "Online" is the parent of "Username", "Password" and "2FA". We first look at the nodes "Username" and "Password", since they are children of the same type. They are connected by an AND refinement, which is represented by a $\wedge$ operator. This results in $\wedge^{p}($"Username", $\vee^{p}($"Phishing", "Key Logger"$))$. "2FA" is a counter child node, and if we look at the transformation Table 1, the structure of a parent with (a) child(ren) node(s) of the same type and one counter child node looks like $c^{p}(\wedge^{p}($"Username", $\vee^{p}($"Phishing", "Key Logger"$))$, $c^{o}($"2FA", "Malware"$))$. The ADTerm looks like this:

$$\wedge^{p}(\text{"Card"}, \vee^{p}(c^{p}(\text{"Find Note"}, \text{"Memorize"}), \text{"Force"})), c^{p}(\wedge^{p}(\text{"Username"}, \vee^{p}(\text{"Phishing"}, \text{"Key Logger"})), c^{o}(\text{"2FA"}, \text{"Malware"}))$$

Finally, on our highest level of abstraction, we add our root node "Bank Account", which has an OR refinement. Our final ADTerm t looks like the following:

$$t = \vee^{p}(\wedge^{p}(\text{"Card"}, \vee^{p}(c^{p}(\text{"Find Note"}, \text{"Memorize"}), \text{"Force"})), c^{p}(\wedge^{p}(\text{"Username"}, \vee^{p}(\text{"Phishing"}, \text{"Key Logger"})), c^{o}(\text{"2FA"}, \text{"Malware"})))$$

With this ADTerm, we can perform more mathematical operations, like determining satisfiability of an ADTerm and determining equivalence between two ADTerms.

A.4 ADTerms Comparison Using Multiset Semantics

Let ADTerm t_1 be defined by:

$$t_1 = (\vee^{p}(\vee^{p}(\text{"Action \#1", "Action \#2}), \wedge^{p}(\text{"Action \#3", "Action \#4"})))$$

and let ADTerm t_2 be defined by:

$$t_2 = (\vee^{p}(\wedge^{p}(\text{"Action \#1", "Action \#2}), \vee^{p}(\text{"Action \#3", "Action \#4"})))$$

and let ADTerm t_3 be defined by:

$$t_3 = (\vee^{p}(\wedge^{p}(\text{"Action \#4", "Action \#3}), \vee^{p}(\text{"Action \#1", "Action \#2"})))$$

The multiset semantics of these ADTerms, based on the multiset model by Kordy et al. [2], are:

$$t_1 = \{(\{\texttt{"Action \#1"}\}, \emptyset), (\{\texttt{"Action \#2"}\}, \emptyset), (\{\texttt{"Action \#3"}, \texttt{"Action \#4"}\}, \emptyset)\}$$

and:

$$t_2 = \{(\{\texttt{"Action \#1"}, \texttt{"Action \#2"}\}, \emptyset), (\{\texttt{"Action \#3"}\}, \emptyset), (\{\texttt{"Action \#4"}\}, \emptyset)\}$$

and:

$$t_3 = \{(\{\texttt{"Action \#4"}, \texttt{"Action \#3"}\}, \emptyset), (\{\texttt{"Action \#1"}\}, \emptyset), (\{\texttt{"Action \#2"}\}, \emptyset)\}$$

Comparing the multiset semantics of t_1 to t_2, we see that, taking into account twin nodes and not taking into account order, these two multiset semantics are not equivalent. Therefore, ADTerms t_1 and t_2 are not equivalent.

Comparing the multiset semantics of t_1 to t_3, we see that, taking into account twin nodes and not taking into account order, these two multiset semantics are equivalent. Therefore, ADTerms t_1 and t_3 are equivalent.

A.5 Levenshtein Distance

Let string 1 be `"Find Note"`, string 2 be `"Find Notes"` and string 3 be `"Finding Notes"`.

We firstly compare string 1 and string 2:

```
"Find Note"

"Find Notes"
```

We determine there is one insertion needed to change string 1 into string 2.

We then compare string 1 and string 3:

```
"Find Note"

"Finding Notes"
```

We determine there are four insertions needed to change string 1 into string 3.

A.6 Satisfiability of an ADTerm using Propositional Semantics: Satisfiable

Given the following ADT from Figure 1, we can determine whether it is satisfiable or not. We have an ADT with an attack root node. When we translate this ADT to an ADTerm, we end up with ADTerms t:

$$t = \vee^{\mathrm{p}} (\wedge^{\mathrm{p}}(\texttt{"Card"}, \vee^{\mathrm{p}}(\mathrm{c}^{\mathrm{p}}(\texttt{"Find Note"}, \texttt{"Memorize"}), \texttt{"Force"})), \mathrm{c}^{\mathrm{p}}(\wedge^{\mathrm{p}}(\texttt{"Username"}, \vee^{\mathrm{p}} (\texttt{"Phishing"}, \texttt{"Key Logger"})), \mathrm{c}^{\mathrm{o}}(\texttt{"2FA"}, \texttt{"Malware"})))$$

The process of building up this ADTerm can be found in Appendix A.3. To determine whether the ADT this ADTerm represent is satisfiable, we firstly determine the propositional semantics of this ADTerm:

$$t = (x_{\texttt{"Card"}} \wedge ((x_{\texttt{"Find Note"}} \wedge \neg x_{\texttt{"Memorize"}}) \vee x_{\texttt{"Force"}})) \vee (x_{\texttt{"Username"}} \wedge (x_{\texttt{"Phishing"}} \vee x_{\texttt{"Key Logger"}}) \wedge \neg(x_{\texttt{"2FA"}} \wedge \neg x_{\texttt{"Malware"}}))$$

We then give all basic actions the value 1, since we assume all basic actions are satisfiable. With this information, we can determine satisfiability of this ADTerm:

$$\begin{aligned}
&\texttt{sat}((1 \wedge ((1 \wedge \neg 1) \vee 1)) \vee (1 \wedge (1 \vee 1) \wedge \neg(1 \wedge \neg 1)))\\
&\texttt{sat}((1 \wedge ((1 \wedge 0) \vee 1)) \vee (1 \wedge (1 \vee 1) \wedge \neg(1 \wedge 0)))\\
&\texttt{sat}((1 \wedge (0 \vee 1)) \vee (1 \wedge 1 \wedge \neg(0)))\\
&\texttt{sat}((1 \wedge 1) \vee (1 \wedge 1 \wedge 1))\\
&\texttt{sat}(1 \vee 1)\\
&\texttt{sat}(1) = 1
\end{aligned}$$

We end up with the value 1, meaning that this ADT is satisfiable and that therefore, the proponent is the winner.

A.7 Satisfiability of an ADTerm using Propositional Semantics: Not Satisfiable

Given the following ADT from Figure 2, we can determine whether it is satisfiable or not. We have an ADT with a defense root node. When we translate this ADT to ADTerms, we end up with ADTerms t:

$$t = \wedge^{\mathrm{p}} (\wedge^{\mathrm{p}}(\mathrm{c}^{\mathrm{p}}(\texttt{"Security"}, \texttt{"During Lunch Break"}), \mathrm{c}^{\mathrm{p}}(\texttt{"Educated Employees"}, \texttt{"Insider Threat"})), \wedge^{\mathrm{p}}(\mathrm{c}^{\mathrm{p}}(\texttt{"Encrypted Data"}, \texttt{"Have Decryption Key"}), \mathrm{c}^{\mathrm{p}}(\texttt{"Strong Firewall"}, \texttt{"Find a Weakness"})))$$

To determine whether the ADT these ADTerms represent is satisfiable, we firstly determine the propositional semantics of this ADTerm:

$$t = ((x_{\texttt{"Security"}} \wedge \neg x_{\texttt{"During Lunch Break"}}) \wedge (x_{\texttt{"Educated Employees"}} \wedge \neg x_{\texttt{"Insider Threats"}})) \wedge ((x_{\texttt{"Encrypted Data"}} \wedge \neg x_{\texttt{"Have Decryption Key"}}) \wedge (x_{\texttt{"Strong Firewall"}} \wedge \neg x_{\texttt{"Find a Weakness"}}))$$

We then give all basic actions the value 1, since we assume all basic actions are satisfiable. With this information, we can determine satisfiability of this ADTerm:

$$\begin{aligned}
&\texttt{sat}(((1 \wedge \neg 1) \wedge (1 \wedge \neg 1)) \wedge ((1 \wedge \neg 1) \wedge (1 \wedge \neg 1)))\\
&\texttt{sat}(((1 \wedge 0) \wedge (1 \wedge 0)) \wedge ((1 \wedge 0) \wedge (1 \wedge 0)))\\
&\texttt{sat}((0 \wedge 0) \wedge (0 \wedge 0))\\
&\texttt{sat}(0 \wedge 0)\\
&\texttt{sat}(0) = 0
\end{aligned}$$

We end up with the value 0, meaning that this ADT is not satisfiable and that therefore, the opponent is the winner.