Universiteit Leiden
The Netherlands

# Opleiding Informatica

# Attack Defence Trees in JSON: Design and Conversion

Steven van Popele

Supervisors:
Nathan Daniel Schiele & Dr. Olga Gadyatskaya

BACHELOR THESIS

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

January 30, 2026

**Abstract**

This thesis aims to improve the usability of attack-defense trees by defining them in a JSON format instead of the current XML format. Attack-defense trees are used to visualize and analyze complex security systems. The work in this thesis is part of a larger project focused building a tool that can create and manipulate attack-defense trees. Specifically, the component presented makes a bidirectional conversion between the JSON and XML formats possible. The current standard of attack-defense trees in the XML format lacks flexibility when manipulating the trees, because it is difficult to traverse and find specific nodes. The sub-goals are to define the format of an attack-defense tree in JSON and to build the conversion between the defined JSON format and the XML format. All the code related to this project is available in the following repository [AIJ].

# Contents

**1 Introduction** **1**

**2 Background** **3**
2.1 Attack-Defense Trees . . . 3
2.2 File Formats for Data Storage . . . 4
2.2.1 Structure and Syntax of XML . . . 5
2.2.2 Structure and Syntax of JSON . . . 5
2.3 Use Cases of XML and JSON . . . 7

**3 Related Work** **8**

**4 Methods** **10**
4.1 Programming Language for Implementation . . . 10
4.2 Evaluating the Conversion Algorithm . . . 10
4.3 Defining Formats . . . 10
4.3.1 ADTs in XML Format . . . 11
4.3.2 ADTs in JSON Format . . . 11
4.3.3 Comparing the Formats . . . 13
4.4 Defining the Experiments . . . 13
4.4.1 Comparing Files Sizes . . . 13
4.4.2 Comparing Performance . . . 14

**5 Implementation** **16**
5.1 Node Structure . . . 16
5.2 Functions . . . 17
5.2.1 build_json() . . . 17
5.2.2 build_xml() . . . 18

**6 Results** **19**
6.1 Comparing JSON to XML . . . 19
6.2 Performance Comparison of the Conversion Tool and Python Module . . . 23

**7 Discussion** **30**
7.1 Definition of an ADT in JSON . . . . . 30
7.2 Evaluating the Formats . . . . . 30
7.3 Evaluating the Conversion Algorithm . . . . . 31

**8 Limitations, Conclusion and Future Work** **32**
8.1 Limitations . . . . . 32
8.2 Conclusion . . . . . 32
8.3 Future Work . . . . . 33

**References** **38**

**9 Appendix** **39**
A Data Format Examples . . . . . 39
B Results of the Experiments . . . . . 43
B.1 Results for Data Format Comparison . . . . . 43
B.2 Results for Algorithm Evaluation . . . . . 45

# 1 Introduction

In the past twenty years, computer science has changed considerably, especially in how data is structured, shared, and understood between systems. Although these advances have resulted in positive outcomes, they have simultaneously led to an increase in cyberattacks [GWM17]. Apart from the growing frequency of these attacks, their sophistication has also increased [BKS11]. Consequently, it is crucial that this threat is addressed by making advancements in cybersecurity. To support the development of these advances, it can be useful to visualize the problems we are addressing. In recent years, researchers have created and improved the attack-defense tree, to support this visualization in a systematic manner. The attack-defense tree (ADT) is a tool to visualize and analyze complex security systems. It visualizes the security system in a way that makes it clear what the overall goal is of an attack and which steps are necessary to achieve that goal. The improvement of ADTs, compared to their predecessor, the attack tree (AT), is their ability to also show possible defenses for individual attack steps [BKS11]. To help spread the use of ADTs and to make the use of ADTs more intuitive, a tool is being developed to improve the accessibility of ADTs. In this thesis, we describe a part of that tool. In particular, we provide a thorough description of the data structures that the tool will provide, and the conversion between these data structures. Specifically, we discuss the XML and JSON data structures. In this thesis, we explain why this conversion is necessary. The goal of this thesis is to create a conversion tool that is capable of defining ADTs in XML and JSON. Additionally, this thesis discusses how the created algorithm for the conversion tool compares to existing solutions to the same problem.

To achieve our goal, we have defined two research questions.

1. How can we define ADTs in a JSON format?

This question is necessary because there is not yet a standard method for defining ADTs in JSON. This is the case for XML, as we will cover in a subsequent section. By defining a JSON format for ADTs, we can make it possible to convert to and from the JSON format. This will improve the usability of ADTs, as users will be able to use and exchange ADTs in different formats.

2. How can we define a method that bidirectionally converts ADTs between XML and JSON efficiently?

   2.1. How does JSON compare to XML, more specifically, how do the ADT relevant features of XML and JSON compare?

   2.2. How well does the conversion algorithm compare to an existing solution, in terms of processing speed?

Question 2 addresses the wider goal of this thesis: the conversion tool. In order to further improve the usability of ADTs, we have to define a method that can perform a bidirectional conversion between XML and JSON.

Question 2.1, which is a sub question of Question 2, will answer why JSON is used and how it can be converted to and from XML. To determine if the method performs adequately compared to existing solutions we will use Question 2.2 as a sub question of Question 2.

To answer these questions, there are several sections that discuss all the necessary topics involved. After this introduction, Section 2 and Section 3 will provide information to help build an understanding of the current state of ADTs and other relevant topics. Section 2 will explain how ADTs work and what the XML and JSON formats are. Section 3 will be used to talk about research relevant to this thesis. In Section 4 we explain our approach to the research questions, including information about the algorithm used to compare the created tool and a description of the experiments. Section 5 describes how we implemented the conversion tool and integrated it into the overarching project in a technical manner. In Section 6 and Section 7 we will present and analyze the results of the experiments and answer the research questions. Finally, we will use Section 8 to review the progress of this project and discuss possible improvements.

# 2 Background

This section provides the foundational concepts on which this thesis is based. The subsections in this section will describe how ADTs work and the file formats we will use.

## 2.1 Attack-Defense Trees

The concept of Attack Trees was created by Bruce Schneier in 1999. Schneier described ATs as a formal and methodical way of describing security systems, based on attacks [Sch99]. In his paper, Schneier defined ATs and gave examples of what they should look like. The paper offers a broad overview of several approaches for modeling security systems. To further establish the concept of ATs, Mauw and Oostdijk published a paper that contains a formal description of ATs that consists of a framework for constructing ATs [MO06]. This paper defines a foundation for ATs and is used as the basis for additional research on ATs.

A visualization of an AT is shown in Figure 1. In the figure the attackers goal is to get free lunch. This goal is split up into three possible sub-goals that can make the main goal happen. The arch between multiple lines signals that all sub-goals with that arch need to be achieved for the parent goal to be successful.

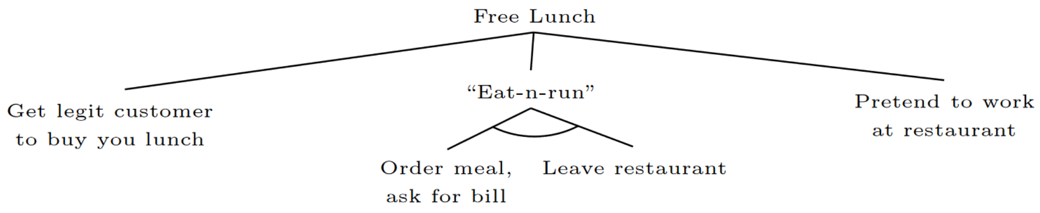
Free Lunch
Get legit customer to buy you lunch
"Eat-n-run"
Pretend to work at restaurant
Order meal, ask for bill
Leave restaurant

Figure 1: Example of an AT [MO06]

In a subsequent paper, Kordy et al. further extended the ATs by giving a formal description of how defensive measures could be implemented into ATs and thus defining ADTs [BKS11]. The previously mentioned paper is used as a basis for this paper [MO06]. The paper extends the formal definition of ATs with the formal definition of ADTs.

The ADT is defined as a rooted tree with labeled nodes that describes measures that an attacker can take to achieve a certain goal. The tree also contains countermeasures that a defender can take to counter these actions. Each node is used to describe these measures and countermeasures. To achieve this, there are multiple types of nodes: the root node, which defines the goal of the attacker; attack nodes, which show the measures to achieve the goal; defense nodes, which show the countermeasures to defend against certain attack nodes. Nodes can have multiple children, and the relationship between parent and child depends on the type of both nodes. For example, when an attack node has a defense node as a child, the defense node is a countermeasure to the attack node. The same is applicable for an attack node as a child of a defense node. It is important to note that a node may have exactly one child of the opposite type. Adding more than one child of the opposite type would lead to a more complicated tree. If the parent and child are the same type, the child is considered a sub-goal of the parent. Children can also have parameters. These parameters are used to give more information about the measures and countermeasures. A node can have two kinds of refinements: a conjunctive and a disjunctive. A conjunctive refinement signals that the actions of all children of a node must be fulfilled to achieve the parent node. A disjunctive refinement signals that only one of the actions of the children must be fulfilled to achieve the parent node [BKS11].

A visualization of an ADT is shown in Figure 2. In the figure the attack nodes are visualized as red circles and the defense nodes as green rectangles. The lines signal a parent-child relationship. Countermeasures are shown with dotted lines and sub-goals with continuous lines. The refinement is shown with an arch. An arch means that the refinement is conjunctive and without an arch the refinement is disjunctive. Parameters could be visualized next to their corresponding branches in the tree, depending on their specific meaning.

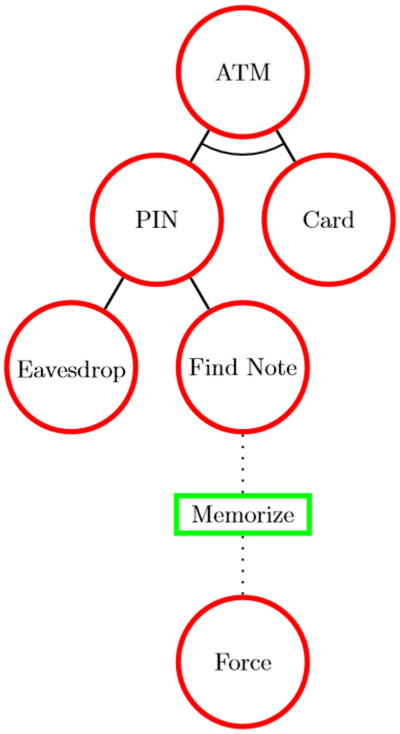

Figure 2: Example of an ADT [BKS11]

## 2.2 File Formats for Data Storage

An important aspect of this thesis is the conversion between XML and JSON. Both are file formats that can be used to store data and are mainly used in web development [XMLa][JSO]. Having different kinds of file formats is useful, because different types of data require different storage types. Certain types of data can be more efficiently saved in one format compared to others. XML, for example, usually uses more bytes to represent the same information than JSON, making it less-storage efficient in many situations [ZUHH15]. How the data will be used is also an important consideration when choosing file formats, since some file formats can be more easily read by humans, while others are optimized for machines [HRD]. For these reasons, it is important for developers to choose the right file format based on the type of data they are using and the intended use of the data. Both formats are text-based, which makes adding parsing functionalities in most programming languages relatively easy.

### 2.2.1 Structure and Syntax of XML

XML stands for extensible markup language. It was originally created to support publishing on computers; however, XML is now commonly used for data exchange on the internet as well [XMLa]. XML can be interpreted by most programming languages [XPM][PSX][LiL].

An XML file is defined by symbols that signal a certain structure. An XML file starts with a line that declares that the file is an XML file. This line includes the version of XML that is used and the encoding of the file. In the first listing in Figure 14 we can see an example of this at line 1. After this line the XML file content begins. The XML file consists of elements defined by tags. Each element contains a piece of the file's content. Elements can be nested within other elements, this is done to provide additional information about the relationship between elements. The manner in which elements are nested defines that relationship. This relationship can be interpreted in different ways. An example of a relationship is a child-parent relationship, where the nested elements are the children of the element in which they are nested. The beginning and ending of an element are defined by tags. There are two types of tags: start-tags and end-tags. The beginning of a tag is signaled by a '<' symbol and the end of a tag is signaled by a '>' symbol. The start-tag of an element contains the name of the element and its attributes. In the XML in Figure 14 at line 3 we can see an example of a start-tag. The name of the element in this line is 'book' and the attributes are 'id' and 'genre'. The end-tag of the example element can be found at line 11. The end-tag of an element consists of the name of the element preceded by a '/'. Between the start-tag and the end-tag of an element we can find the content of the element. Content can be text, other elements or a combination of both [XMLb].

### 2.2.2 Structure and Syntax of JSON

JSON stands for JavaScript Object Notation. JSON is a file format that was created to be easily processed by both computers and humans, and therefore is considered to be lightweight [JSO]. The format is based on JavaScript objects, hence the name. Because of this, JSON is highly functional in JavaScript. Apart from JavaScript, many other programming languages can process JSON [JED][JJM][LiL]. JSON is gaining popularity in many fields of computer science, which makes it more attractive for developers to use JSON as a file format and to create functionality within programming languages to process JSON [TLH18b].

JSON can be seen as a combination of an array and a dictionary. JSON changes its behavior based on different symbols. The dictionary form uses '{' to signal the start of a dictionary object, '}' to signal the end of a dictionary object and ':' to assign values within the dictionaries to the corresponding keys. JSON uses '[' and ']' to indicate the starting and closing points of an array [TLH18b]. A dictionary can have an array as value of a key, as can be seen in the second listing of Figure 14.

To illustrate the difference between XML and JSON we have put the same data into both structures and put them side by side. Later on in this thesis we will go into the differences of these two formats.

```xml
<?xml version="1.0" encoding="UTF-8
  "?>
<library>
  <book id="b1" genre="fantasy">
    <title lang="en">The Adventure
       </title>
    <author>Someone</author>
  </book>

  <book id="b2" genre="sci-fi">
    <title lang="en">Journey to the
        Stars</title>
    <author>Someone Else</author>
  </book>
</library>
```

XML

```json
{
  "library": {
    "book": [
      {
        "title": {
          "lang": "en",
          "text": "The Adventure"
        },
        "author": "Someone",
        "id": "b1",
        "genre": "fantasy"
      },
      {
        "title": {
          "lang": "en",
          "text": "Journey to the
              Stars"
        },
        "author": "Someone Else",
        "id": "b2",
        "genre": "sci-fi"
      }
    ]
  }
}
```

JSON

Figure 3: Illustration of the same data encoded in XML and JSON.

## 2.3 Use Cases of XML and JSON

Each format offers advantages in different scenarios; we will explore some use cases and identify which format is most suitable in each of them.

Communication between systems that are not locally connected is done via web APIs. Web APIs communicate with structured data. Both JSON and XML can be used for this and are used often. JSON is often chosen over XML, because JavaScript is often used in the front-end of most projects that use web APIs. A JSON response usually contains less text than XML, which improves compatibility [WAD]. This also means that JSON files have less bytes than XML files, which improves transmission efficiency [ZUHH15]. Additionally, XML is often implemented with the use of a specific, which has a strict format and is thus not universally understandable [SOP].

XML has great functionality for storing documents. It is able to store both the contents of a document and the metadata for documents and make a distinction within the format [XMLa][ZUHH15]. In contrast, JSON lacks native mechanisms for this distinction, as it cannot make distinctions between different types of attributes [JSO].

XML and JSON are useful for configuration files, because of the way that both formats are structured [B17][Car18]. Configuration files are often used to setup a program and provide it with its initial values. An advantage of JSON for this is that the values can be easily found and edited by a human reader [JSO]. XML on the other hand can handle configuration files with metadata more effectively, as we have seen before.

Many programs make use of log files to record their runtime. Logs in JSON format are structured so that can be easily parsed by an analysis program and they are readable by humans [JSO]. In this case XML would be too word heavy and difficult to parse.

# 3 Related Work

In addition to Section 2, this section expands on the research done on ADTs. We discuss research that is similar to ours and show its connection. The topics will be divided into tooling methods, JSON structures, comparative studies of XML and JSON and conversion algorithms.

The overarching project that this thesis will be part of is mainly concerned with providing a visualization tool for ADTs. Currently, ADTool is a widely used application for constructing and visualizing ADTs. It provides the means for modeling and displaying ADTs. In the ADTool manual, Kordy et al. write that the goal of ADTool is to provide a graphical and user-friendly way to work with ADTs, as well as to give users tools to conduct a thorough analysis of their trees [BKS13]. ADTool makes use of XML files to save ADTs. As the field of ADTs and risk assessment has evolved a new tool was created by Gadyatskaya et al., ADTool 2.0. This tool is significantly more advanced than its predecessor. This enables the tool to offer multiple new features. In addition to new features, like scripting, copy-pasting within trees and analyzing large trees, the tool also provides support for more advanced ADTs and node prioritization. Notably, ADTool 2.0 also supports the ability to accept XML input files that are not written in the ADTool XML schema [OGTR16]. This last feature can significantly improve the usability of the tool. In this thesis, we use the created XML format of ADTool to define a JSON version and with it we aim to further enhance the usability of ADTs. A tool with a similar purpose as the ADTool variants was developed by Weiser and Delcourt. In their work, they built a tool for modeling ADTs. This tool is also capable of handling JSON files. However, it is not capable of making conversions to other file formats, like XML [TWL22]. Our conversion tool aims to add additional usability. While a JSON definition is given in their work, it lacks the compatibility with XML needed for the tool we are developing.

In the following paragraph, we examine how previous research has approached representing data in JSON. The research of Weiser and Delcourt contains another important feature, namely, their choice of using JSON to represent ADTs. They use an altered version of the definition from Kordy et al. made by Wideł et al.. Wideł et al. extend ADTs by implementing refinement and countermeasures as individual nodes [BKS11][WWP14][TWL22]. This adds an extra layer of abstraction, which in turn increases the definition's complexity. On top of that, the way Weiser and Delcourt treat nodes makes the use of parameters difficult. In order to improve usability we want to be able to add parameters and we need a simplified definition. In their paper, Jhawar et al. introduce a mathematical approach to a sequential conjunctive refinement. This refinement adds the ability to formally define SAND relationships between nodes [RJRR15]. While this addition extends the usability of ADTs, it simultaneously introduces complexity and incompatibility with other formats. The paper by Schiele and Gadyatskaya describes a method for comparing attack trees. To perform this comparison, the authors define a metric that uses important attributes of the trees [SG25]. Based on the attributes used by the metric, we can deduce what important attributes are that each node must have. The research done by Lv et al. surveys the ways that data can be saved in JSON. The paper discusses several approaches to writing JSON objects and retrieving data from them. Lv et al. make use of *recursion*, in which a certain structure is repeated within the JSON object to efficiently retrieve information [TLH18a]. The JSON objects constructed in this thesis will also employ recursion, as the structure of every node is the same. This will ensure consistency between nodes, which in turn makes the ADT more traversable and adjustable.

There are many ways of storing data and to determine what every format excels in many comparisons have been made between these formats. We are going to look at a few comparisons between XML and JSON, to see how these studies have approached these comparisons. In the paper written by Zunke and D'Souza a comparison is made between XML and JSON. They measure the comparison of the two formats in features and in performance. They conclude that JSON achieves a better performance compared to XML. However, because of the structure of XML, there are certain situations where XML can offer more flexibility in saving data. The structure of XML lets you describe any kind of data, where JSON is limited to a set number of data types [ZD14]. Due to the nature of this thesis, we can disregard the visual aspects of the two formats. However, this thesis benefits from the performance advantages of JSON, as it is the main file format we use. Nurseitov et al. have done a study to see which data interchange format, XML or JSON, is the better choice. The study concludes that JSON is significantly faster in use and uses less resources when used for data transfer. They found this result by doing a combination of experiments, including the timing of transferring objects and the amount of computer resource utilization. They used different scenarios using different properties [NNI09]. In the overarching project the ADTs will be transferred to different systems, so even though data interchange is not directly relevant to this thesis, the results are positive nonetheless. On top of that, their results also show that JSON performs better outside of data interchange. We can conclude that processing JSON strains computer resources less than XML, which is important for modifying data.

Converting XML to JSON and back has been tackled by multiple tools before. These tools might not fit our use case, but can be used as inspiration or be incorporated. The tools in question will be covered in this paragraph. The next paper we will discuss focuses on converting XML data to different data representation. To do this conversion Martens et al. make use of mapping. To convert the XML data they try to find parts of elements that are recognized by a mapping document. With the help of this document the elements can be mapped towards the corresponding instances of the other data form [DVDdW08]. In this paper we can see a situation where XML is not directly converted to the desired format. Our conversion will mimic this idea by using an intermediary data structure instead of mapping. Šandrih et al. propose an efficient and unified approach to converting between XML and JSON. In their paper they created an overview of conversions between XML and JSON. These conversions are the result of using two different converters [B17]. Their results are not applicable to our project, because we will mimic a specific ADT format in XML in JSON. Therefore a custom conversion process will be developed. The JavaScript library xml-js was written to create a bidirectional tool converting XML to JSON and back. While other libraries did exist, like the DOM parser and the xml2json parser, their output merges nodes together in a compact manner. Nashwaan created xml-js to preserve the order of elements from XML in JSON [Agu][Nas][DOMa]. Our conversion tool needs to do both, and preserve order and be compact. The attack sequence is lost without the order, so we need a tool that can keep this order. However, we also want the output of this tool to be compact, because this makes the JSON more traversable and thus more modifiable. The DOM parser does have a useful feature, namely, we can use it for creating XML objects out of XML strings [DOMa]. In this way, it can be used in our conversion from JSON to XML to build an XML object. This object can later be used to improve compatibility, by passing it on to the overarching project, where it can be returned to the user.

# 4 Methods

This section describes the methods used to create and evaluate the conversion tool and to run the experiments. Important aspects of this section are the algorithm to compare our tool with and how data structures are used.

## 4.1 Programming Language for Implementation

A key design decision in this thesis is to write the conversion tool in JavaScript. The motivation for this choice is that the project that this tool will be part of is written in JavaScript as well. Writing in JavaScript ensures compatibility with the rest of the project, as there are no extra steps needed to communicate between different JavaScript files [GLO]. Additionally, JSON is derived from JavaScript objects, therefore using JavaScript is a natural choice. [JSO].

## 4.2 Evaluating the Conversion Algorithm

In order to say something about the performance of the conversion algorithm developed for this thesis, we will need to compare it with another similar algorithm. We will measure the processing speed of our algorithm and we will compare this speed with the speed of using an algorithm from an existing Python module. To do this, we have rewritten the JavaScript code in Python. Python will be used to experiment with, because it is able to read and write files directly from and to the working directory. This will be used in the experiments conducted in this thesis. Accessing local files in JavaScript requires the use of extra steps via modules or APIs [RFN][FSA]. The implementation section will describe how the conversion algorithm works. In this subsection, we will describe the Python module that is used.

The xmltodict module is used to convert XML to JSON. Python has two datatypes that can be used to represent a JSON object, these are lists and dictionaries. To convert XML to these two datatypes xmltodict makes use of an expat based XML parser [Ble]. Expat is a fast XML parser that can be found in various XML parser implementations across different programming languages [Zha07]. The speed of expat can be attributed to the fact that an XML file is processed in multiple pieces. This ensures that parts of the XML can be parsed before the complete document is received [Coo]. We will only use the class-internal function *xmltodict.parse()*. This function does the conversion and creates dictionary keys for all XML elements. It treats XML elements differently than we will do. For example, with nested elements of the same type, it creates a list of that type instead of a dictionary.

## 4.3 Defining Formats

For the conversion of the ADTs we need to define what the XML and JSON representations of the ADTs look like.

### 4.3.1 ADTs in XML Format

For XML, we use the definition of ADTs as given in the ADTool manual. We do this to ensure that this project is compatible with ADTs designed in the ADTool application. This is an important design choice, as it ensures that users can use their already existing ADTs in the new tool.

The tree is defined inside an adtree start- and end-tag. The nodes of the tree are represented by node tags. The start-tag of these node tags include two attributes. These are the refinement and the role the node has relative to its parent, called 'switchRole'. A node is closed by an end-tag. Every other component of the node is defined between the start-tag and end-tag of the node tag. The name of the node is represented by a label tag. The name itself can be found between the start-tag and end-tag of a label. The parameters of the node are represented by parameter tags. Inside the parameter start-tag there is a domainId attribute that represents the name of the parameter. The value of the parameter can be found between the start-tag and end-tag of a parameter [BKS13]. An example of an ADT in XML is shown below. The example is a trimmed down version of the XML in Figure 13 from the ADTool manual [BKS13]. The full figure can be found in appendix A In Listing 1, we can see how nodes are nested within the adtree element and within other nodes. This is shown by the positions of the start- and end-tags of the element. From the example we can also see that the 'switchRole' attribute is only present when a node switches role. The same is true for parameter elements, these only exist when a node has parameters. The parent of the node on line 12 is the node on line 5. From the 'switchRole' attribute of this node we learn that it is a countermeasure for their parent.

```xml
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <adtree>
 3     <node refinement="conjunctive">
 4         <label>steal money</label>
 5         <node refinement="disjunctive">
 6             <label>learn PIN</label>
 7             <node refinement="disjunctive">
 8                 <label>find PIN</label>
 9                 <parameter domainId="MinTimeSeq1">100.0</parameter>
10                 <parameter domainId="ProbSucc2">0.1</parameter>
11             </node>
12             <node refinement="conjunctive" switchRole="yes">
13                 <label>security training</label>
14             </node>
15         </node>
16     </node>
17 </adtree>
```

Listing 1: Example XML ADT

### 4.3.2 ADTs in JSON Format

For the JSON example we have to convert the XML tags to JSON objects. In the XML definition there are a few features that we need to implement in the JSON definition. These features are essential for accurately reproducing the intended ADT.

In Listing 2 we show a JSON that shows the definition of all the features of ADTs as they will be used in this thesis. The root of the JSON is the adtree object. Every node is an object on its own, within the adtree. In the Listing 2 we can see that every node is denoted by a number. This number corresponds with the order of the nodes from left to right. The adtree object behaves as a node and is therefore implemented as one. The adtree and node objects have a label, a refinement, a role, children and parameters. These all come from the XML definition. The label, refinement and role are keys with values. The node objects are implemented as children objects of the adtree. They are put in their corresponding parent. This can be seen on line 6 and line 21 of Listing 2. The parameters array in a node consists of separate parameter dictionaries with a name and a value. The names of the parameters are the keys in the dictionary. The adtree object and the child objects all have the same structure. This consistency is important to make the JSON traversable. The ADTs that will be created in the tool can vary in size, therefore it is appropriate to design them with a predictable structure. This consistency allows traversal algorithms to operate without prior structural knowledge of the tree. The tree can still be used to find specific values or to change subtrees after being constructed.

```
 1 {
 2     "0": {
 3         "label": "Bank Account",
 4         "refinement": "0",
 5         "switchRole": "0",
 6         "0":{
 7             "label": "ATM",
 8             "refinement": "1",
 9             "switchRole": "0",
10             "parameters": [
11                 "1":{
12                     "name": "MinTimeSeq1",
13                     "value": 20.0
14                 },
15                 "2":{
16                     "name": "ProbSucc2",
17                     "value": 0.25
18                 }
19             ]
20         },
21         "1":{
22             "label": "Online",
23             "refinement": "0",
24             "switchRole": "0",
25             "parameters": []
26         }
27         "parameters": []
28     }
29 }
```

Listing 2: Example JSON ADT

### 4.3.3 Comparing the Formats

For our project we must identify the differences between JSON and XML. We will only examine differences that are relevant to our intended use of the formats. The JSON representation uses more elements than the XML representation. The extra attributes are represented as elements themselves in JSON. The XML representation can leave out unused attributes and elements can differ in the number of attributes present. For JSON this is not the case. We can see that even if a node does not have children and/or parameters the object for them still exists. This is the same for the *switchRole* attribute, even if it has the value *"no"* it still has to exist. In XML end-tags repeat the name of the element that is being closed. These end-tags are equivalent to the "}" symbol in JSON. Another important aspect to consider is their functionality within JavaScript. Both JSON and XML can be used in JavaScript. However, JSON is based on a built-in feature in JavaScript, specifically, JavaScript objects. This means it can be parsed without the need of external functionalities. For example, to parse XML we need to use an external parser, like the DOM parser, to use its format properties [DOMb].

## 4.4 Defining the Experiments

To help answer the research questions two experiments were conducted. The first experiment focuses on the difference in file size between XML and JSON. With this experiment we aim to answer Question 2.1, while the second experiment aims to address Question 2.2. The second experiment will test the efficiency of our implementation.

### 4.4.1 Comparing Files Sizes

For the first experiment we are going to compare the file size of multiple ADTs for the XML and JSON formats. For this we will use ADTs that are vary in height and width. The ADTs take the form of a perfect m-ary tree, where every internal node has exactly m children and all the leaves have the same depth [Nyb14]. With height we mean the maximum path length from root to leaf within the tree and with width we mean the number of child nodes each internal node has. Leaf nodes have by definition no children [MO06]. The ADTs resulting from this will be labeled according to their width and height, so an ADT with a width of 4 and a height of 7 will be named 4×7. In Figure 4 is an illustration that shows how height and width affect the ADT, more specifically what is controlled with either of them. The width in this figure is 2 and the height is 3.

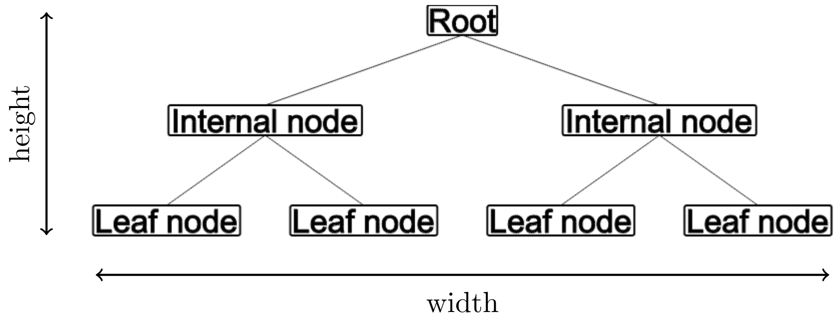

Figure 4: Illustration depicting what is influenced by height and width.

Here width illustrates what the number of children does to the tree, not how the width is defined. The different heights and widths will ensure that we will have ADTs representing diverse sizes and therefore have enough results to be able to draw conclusions. The file size will be measured in bytes. We will do two comparisons; one that shows the file size of ADTs when both width and height increase by one for each next ADT, and one in which we will compare the file sizes for smaller ADTs. The first comparison provides a broad overview for if the size difference holds up for a wide range of ADT sizes. The second comparison will show whether the result of the first comparison is similar for realistic sizes.

### 4.4.2 Comparing Performance

To test the efficiency of the algorithm for the conversion from XML to JSON we can run an experiment that measures the conversion time for a variety of different ADT sizes in seconds. We have chosen to compare different heights and widths of ADTs. For this experiment we use the same definition of width and height as the previous experiment. The resulting time of each size depends on multiple iterations of converting the same tree and taking the average [ŠM14]. To find a suitable number of repetitions we will do the experiment with six different repetition numbers. Each time we increase the repetitions, we expect that the variability in the data to decrease. This will cause the resulting averages to converge to the true value of the process time of the ADTs [Sel15]. We will also run the same experiment on an implementation using xmltodict, a module from a native Python library, to assess how efficient our conversion tool is compared to that implementation [Ble]. To ensure that we are only comparing between the two algorithms and not between Python and JavaScript we will have to rewrite the same code of the conversion tool from JavaScript to Python. Both implementations will be written in Python and both will be tested using Python. This results in two sets of data: one set for the conversion tool and one for the Python module. The experiment will be performed on a system with the following specifications: an Intel Core i5-13600KF CPU, 32 GB of RAM, Windows 11 (version 25H2) and Python 3.9 [DB]. To find the resulting process time of each ADT for both algorithms we utilize Python's datetime module [DT].

The ADTs that are generated for this experiment grow exponentially in size. Therefore, we expect the execution time for these ADTs to also grow exponentially. This means that for some ADTs the execution time might become unreasonable. To prevent this from happening we will make use of a threshold. We have chosen to find a threshold for the execution time of ADTs within our data and cut off the execution of the algorithms for ADTs that exceed this threshold. To find the threshold we will run our conversion algorithm a single time for each ADT. Due to the expected exponential growth we can find that the execution time will rise steeper the larger the ADTs are. Running the conversion algorithm a single time for each ADT does not guarantee reliable data, but it does show a decent enough overview of the growth in execution time. Based on this we can decide for which level of steepness we can cut off the executions. This will be our threshold. To make sure cutting off the executions will not result in the loss of necessary data, we need to choose a sufficiently high threshold to observe growth trends for each algorithm. Furthermore, we will be able to observe if an algorithm is able to process larger ADTs than the other algorithm by looking for which ADT size the execution was cut off, this is important as we are comparing two different algorithms. If the processing time of an ADT exceeds the threshold, we mark it with a '-' in the results table.

# 5 Implementation

There are multiple ways to implement the methods described in the previous section. The code base for this project can be split into various parts. First, the implementation of components responsible for knitting the code together will be discussed, after that we will examine two parts that are necessary for the conversion.

The conversion itself is written in a separate file and is called by the responsible function within the project. This is done to avoid interference with other sub-projects. The code for the conversion only expects two variables. The first variable is an integer that signals which kind of conversion needs to be performed: XML to JSON or JSON to XML. The second variable contains the input. This is either the XML or the JSON that needs to be converted. For both ways of conversion, the input is expected to be a list of strings. There are two extra methods of receiving input that can be handled in the tool. One method is to generate an XML, as a list of strings, within the tool. The other method is to receive the XML from a source outside of the tool. Both these methods were implemented during the course of this thesis. For the implementation of the tool within the overarching project we expect a list of strings as input when the tool is called.

The first method of input is useful for running experiments on the tool. For example, in this thesis we did an experiment to measure the speed of the conversion algorithm. To do this we generated XMLs of multiple different sizes and ran the conversion tool on these XMLs. This method would be too costly in terms of time to be used within the tool and changing the output ADT would not be straightforward. The second method of input was used to test the tool during development. In JavaScript, it is non-trivial to access local files from the computer. Therefore, the XML had to be retrieved from an online source, which is a built-in feature of JavaScript. This meant that we could test the tool without having to manually enter the XML every time or write the XML within the code of the tool. The downside of using this method is that using online sources means that all functions that use the data from the source must be asynchronous. This is because the data is not immediately available, so the functions that use the data must wait for the data to be received [ASY]. The difficulty with asynchronous functions is that the code runs while a part of the data being used has not yet been received. Parts of the code that are dependent on that piece of data must wait for the data to be received. This increases the complexity of the code, making it more difficult to interpret, maintain, and debug.

## 5.1 Node Structure

To assist in converting to JSON, the tool uses an intermediate structure. A direct conversion is hard to achieve, because the location of a node in the tree is not yet known during the creation of a node. The JSON is built as an object and nodes are added to the JSON ADT as objects themselves. Saving the information for each node as an intermediate object makes it possible to add information while building. In the end, all node information is known, this includes their place in the tree. With this information, the nodes can be added to the tree in the right manner.

The node structure is a class containing all the important information that is needed to build the JSON. In the end all this information is saved in the JSON. The label, refinement, switch_role, and parameters variables come from the XML input. The parent, order label and depth are derived during the process of creating nodes. These three variables are necessary for finding out the JSON structure of the ADT. With the parent variable we can determine which nodes are siblings. The order of a node is derived from the number of previously found siblings. The order label is an important attribute of the node class. It is used to save the position of a node within the ADT. The order label of a node consists of a series of numbers and dashes. Every number in an order label is the order of a parent node, and the last number is the order of the current node. An example would be *0-0-1*, in this example the order label represents the second child of the first child of the root node.

## 5.2 Functions

To achieve the conversion we have created two important functions. The *build_json()* function converts the XML input to JSON, and the *build_xml()* function does the reverse. The first of those two was created with minimal use of libraries, while the second relies for the creation of an XML object on a library.

### 5.2.1 build_json()

This function loops through all the lines in the given list of strings. For every line it looks for significant characters. Characters are significant if they signal the start or end of an XML tag and if they indicate the role of the tag. It is important to note that not every line of the XML starts with the same indentation, therefore the first significant character of a line is a '<' as this signals the start of a line. With this in mind it follows that the function will first look for '<' when starting on a new line. After the '<', a character follows that determines the role of the current element. These roles start with one of the following letters: *a*, *n*, *l*, or *p*. The */* signals the end of an XML element. The function filters out tags with specific prefixes, namely, *a*, *l*, and *p*, as these are handled differently. The lines that have a tag starting with *n* are the start of a node element.

The function searches for every node element for the label, the refinement, whether it switches role and the parameters. The label and parameters can be found on the lines with tags starting with *l*, for label, and *p*, for parameters. The refinement and whether it switches role can be found in the same line as the node tag. The four features are retrieved by three functions. These work by iterating over the characters in each line and are written in such a way that they can recognize when each feature starts and stops. This process relies on the standardized format of the XML ADTs. After collecting all features of the node, the outcomes are given as arguments to the *insert* function. It is important to determine whether the current node is the root node or not, as we do not have a parent for the root node. On top of that, the depth, which is equal to the indentation, of the node is also tracked. These properties are necessary track of which node is the parent of which child.

*Insert* adds all the features of the node into the node structure. All information but the parent node and the oder label is saved in the same way for every node. The parent of the root is the root itself. The root node is assigned the order label '0'. The parent of non-root nodes is dependent on the depth of the current node compared to the previous node. There are three distinct situations for this comparison: the current node is deeper than the previous node, less deep, or the same depth. In the first situation the previous node is the parent node of the current node. The order label of the new node is the order label of the parent node plus "-0", as the current node is the first child of the parent. If the previous node is deeper than the current node, it means that the current node is a sibling of the parent of the previous node. The function will loop through the list of added nodes, from last to first, and searches for nodes with the same depth. The parent of the current node is equal to the parent of the first node found with the same depth. The function counts every node that has the same depth and the same parent, this determines the order of the current node as a child is of its parent. The order label of the first node with the same depth will be stripped of its last character and will be replaced by the order of the current node to form the order label of the current node. The last node must have a parent; therefore a previous node with the same depth must exist. If the current node is the same depth as the last node it means that the last node is a sibling of the current node. Therefore, we can take the order label of the last node and increase its last number by one. As the last and current node are siblings they must have the same parent. It returns the fully populated node to the build_json() function. The node is then saved in a list that contains all nodes.

At the end, the build_json() function loops through all the nodes in the list and uses the *to_json()* function to create the JSON ADT. To create a JSON we have to create an object for every node and add its information to the new object. In the function we start with an empty JSON object and for each node we use the order label of the current node to check whethe the parents in the order label of the node already exist in the JSON object. If an intermediate parent node does not exist, the function creates a new parent object. After that, an object is created for the current node and added to its corresponding parent. The JSON ADT is then returned to the project.

### 5.2.2 build_xml()

To build the XML ADT, we first construct a string reflecting the structure of an XML file. After that we use the DOM parser to parse the string into an XML object. For adding children to a node we will use a recursive function that adds all features of a child to the string and adds the children of that child. The XML strings are built as defined in the ADTool manual [BKS13]. Because of the defined JSON format in Appendix A, we can retrieve every feature of a node by using the defined keys. These features are placed in the correct position in the XML string.

# 6 Results

In this section, we will cover the comparison in Subsection 4.3.3 and the results for the experiments described in Subsection 4.4. For the results in this section we will use the same definition of width and height used in Subsection 4.4. The width of an ADT is defined as the number of children each node in the tree has and the height of an ADT is defined as the maximum path length from root to leaf. ADTs will be named using width $\times$ height, so an ADT with a width of 5 and a height of 2 will be named $5 \times 2$.

## 6.1 Comparing JSON to XML

Referring to Subsection 4.3, we find the differences between the JSON and XML versions of ADTs. The two examples in Listing 1 and Listing 2 describe the same ADT. Because they need to convey the same information, we can make a good comparison between the two formats based on their differences.

Because of the existence of unused attributes in the JSON format, changing their values requires less effort. If we want to add an element, we can locate the attribute it belongs to and add its data at that position immediately. In XML we would need to know where this new element should be placed, or in other words, we need to know what the surrounding elements look like and compare that to the XML file at hand. Iterating over these XML files is also harder because XML elements lack a uniform structure, whereas JSON is uniform for the defined format. JSON provides better support than XML in JavaScript environments. The use of an extra parser makes using XML complicated compared to the integrated use of JSON. These things mean that writing modular code for parsing JSON will be easier in JavaScript.

For a human, an end-tag can be argued to be useful [Kha15]. It shows us where an element is closed. With '}' in JSON we know that an element is closed, but it is not necessarily clear which element this is. For XML this is clearer, as the name of the element is repeated in the end-tag. On the other hand, we can also argue that the use of end-tags make the representation more complex, as there is additional text. Another property of both formats is the line count. This count depends on the layout style. If we take a look at Figure 14 we can see the same data being encoded in both XML and JSON. The JSON ADT has 24 lines, whereas the XML ADT has 12 lines. The line count of the JSON ADT is two times the line count of the XML ADT. Initially, this seems to be a negative property of the JSON format. However, we can also see that JSON breaks down the lines of the XML file in multiple parts, which can be argued as to make it more readable for humans. The line count is partially the result of attributes being treated differently in JSON. The *refinement* and *switchRole* attributes of a node are saved in the same manner as the *label* of a node. Line count does not matter for computers. Every line of both formats can be put on a single line and a computer can process it just as easily.

A property that does matter for computers is the file size of both formats. Using the ADTs of varying sizes we can compare the file sizes of the two formats. We will use the ADTs that are described and generated in subsection 4.4.1 for this. We have established above that the computer can use files in either format containing all the information on a single line. If we implement this on the ADTs, we can determine the minimum file size for the ADT in both formats.

First, we will examine ADTs that grow by one for both the height and width for every new ADT. The results of this can be found in Table 2; a corresponding graph is shown below, with the sizes in bytes. The graph uses a logarithmic scale to improve visibility. The file sizes grow exponentially and without this scale this growth would not be visible. The x-axis shows the size of an ADT using width × height.

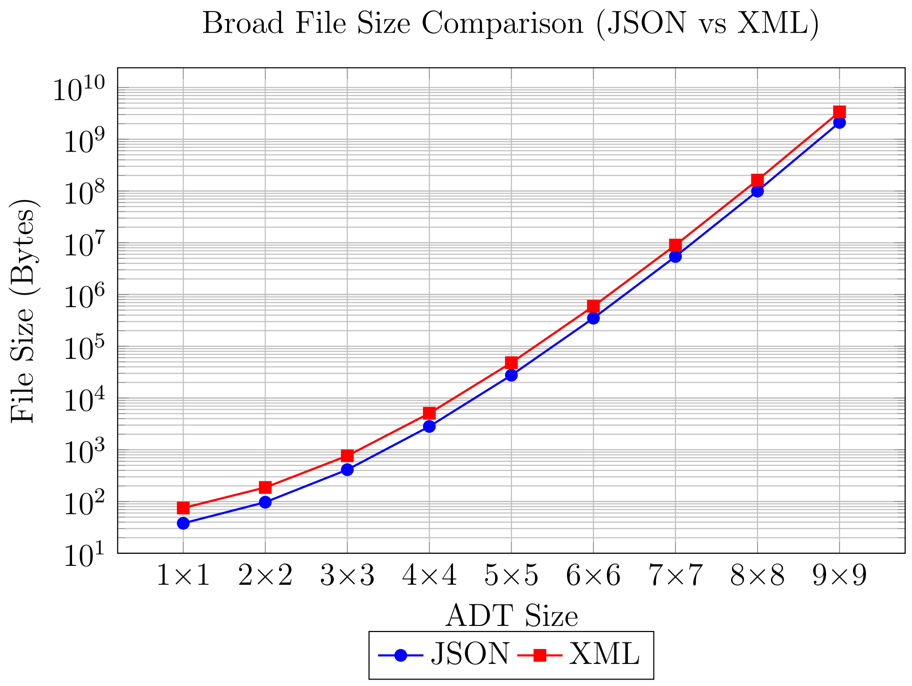

Figure 5: File size growth for JSON and XML formats over increasing ADT sizes on a logarithmic scale.

As shown in Figure 5 the file sizes of ADTs in JSON format are consistently smaller than those in XML format for all tested sizes. For example, using the data in Figure B.1 we can see that an ADT of size 1×1 is 50% smaller in JSON and an ADT of size 9×9 is nearly 63% smaller in JSON format.

Next, we compare smaller ADTs. Full results can be found in Table 3 and a graph can be found below. Again we will use a logarithmic scale to improve visibility. We have plotted five different lines, each representing an increase in height and keeping a consistent width.

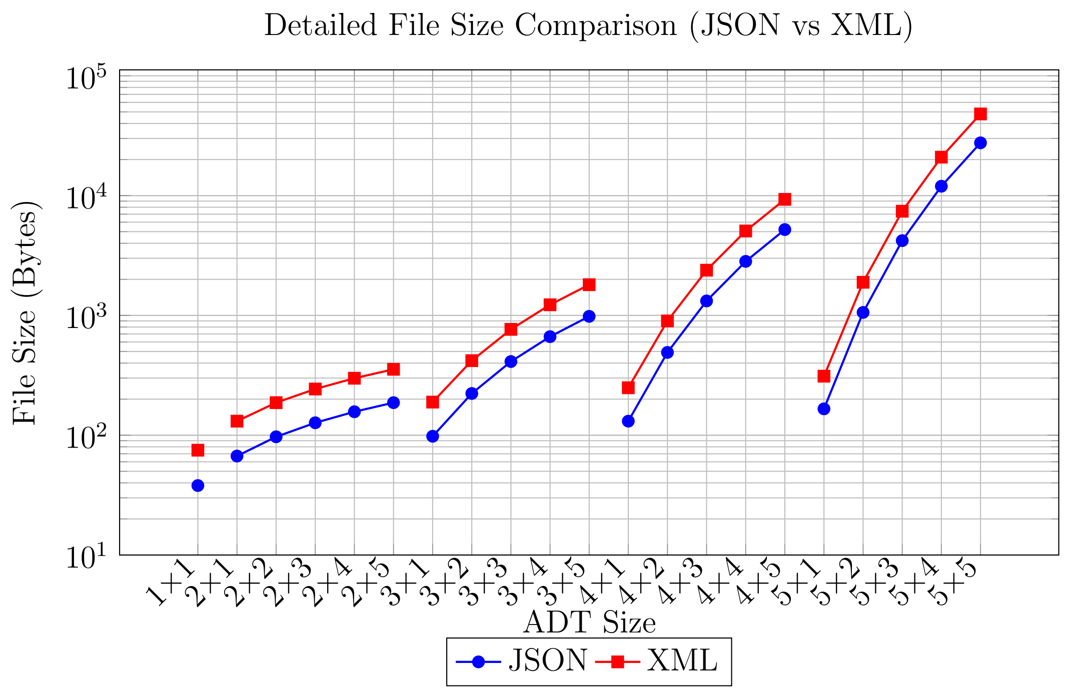

Figure 6: File size comparison for JSON and XML formats across various smaller ADT sizes on a logarithmic scale.

In Figure 6 we can see that the file size of JSON stays smaller than XML for smaller ADTs. An interesting detail we can observe is that the difference between XML and JSON seems to grow roughly logarithmically. This means that in truth the difference between the two lines grows much more radically. This is also supported by the results in Appendix B.1.

In the following graph the file size ratios of both formats have been plotted. The data can be found in Table 5. This ratio is determined by dividing the file size of the larger ADT by the file size of the smaller ADT. With the file size ratio we can gain an understanding of how quickly the file size grows for bigger trees.

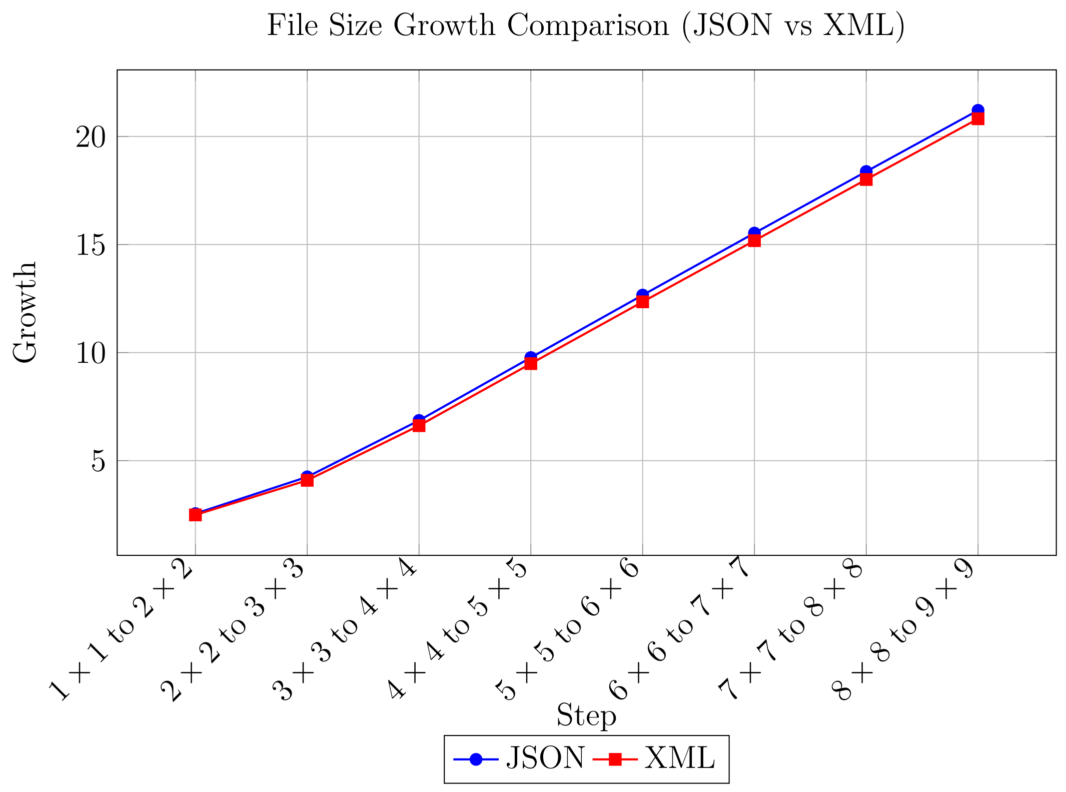

Figure 7: The growth rate of file sizes of tested ADTs.

This graph shows that the two formats grow at nearly the same rate, with a slightly faster growth for JSON with larger trees. For the formats we can calculate the order of magnitude to find the overall growth of both. We do this by dividing the largest file size for each by the smallest for each [KJMT18]. For XML this is

3.377.822.403 / 75 $\approx 4.5 \times 10^7$,

and for JSON this is

2.113.324.966 / 38 $\approx 5.6 \times 10^7$.

From these calculations we can conclude that both grow by roughly seven orders of magnitude for the range we observed them in. This shows that both XML and JSON ADTs scale in a similar range. Although JSON seems to grow slightly faster for larger ADTs. Due to technical limitations we are unable to determine the ADT size at which JSON overtakes XML in file size.

## 6.2 Performance Comparison of the Conversion Tool and Python Module

This subsection consists of the results for finding the setup properties and the results for the performance comparison. We will first describe the results concerning the threshold and repetition number, as described in Subsection 4.4.2. These results come from only using the conversion tool. Afterwards we will give the resulting execution times for the ADTs described in Subsection 4.4 for both the conversion tool and the Python module.

In the table below we can find the execution time for a single run of every ADT using the conversion tool. The executions that will take much longer than would be reasonable have been marked with a ”-”. Results that are notably larger than the previous result in their column and row have been shaded red. This is done to emphasize a steep rise in execution time. Results have been rounded to four decimals [Col15].

| | | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | 1 | 0.0035 | 0.0154 | 0.0157 | 0.0021 | 0.0057 | 0.0038 | 0.0116 | 0.0014 | 0.0144 |
| | 2 | 0.0109 | 0.0104 | 0.0121 | 0.0095 | 0.0136 | 0.0082 | 0.0111 | 0.0110 | 0.0074 |
| | 3 | 0.0054 | 0.0131 | 0.0068 | 0.0112 | 0.0129 | 0.0126 | 0.0113 | 0.0127 | 0.0047 |
| | 4 | 0.0038 | 0.0123 | 0.0050 | 0.0108 | 0.0079 | 0.0129 | 0.0138 | 0.0130 | 0.0126 |
| Height | 5 | 0.0145 | 0.0099 | 0.0121 | 0.0153 | 0.0162 | 0.0362 | 0.0865 | 0.1297 | 0.2841 |
| | 6 | 0.0108 | 0.0079 | 0.0186 | 0.0328 | 0.1241 | 0.5205 | 1.8683 | 5.7672 | 15.9185 |
| | 7 | 0.0009 | 0.0105 | 0.0295 | 0.2562 | 2.4173 | 16.2164 | 85.1991 | 383.3982 | 1364.8383 |
| | 8 | 0.0100 | 0.0128 | 0.1291 | 3.4862 | 57.6046 | 628.5310 | 4414.3664 | - | - |
| | 9 | 0.0074 | 0.0140 | 0.8893 | 55.0767 | 1570.6664 | - | - | - | - |

Table 1: Execution time for converting varying ADT sizes using the conversion tool based on a single run.

We can see a steep rise in execution time in this table. This rise creates a gap in the results between 6 and 15 seconds. We chose a threshold of 8 seconds, which ensures that the threshold is large enough for results before the steep rise to grow and small enough for results after the steep rise to not fit in the threshold after doing more repetitions. For the experiments that will follow in the rest of this section we will cut off the execution of the ADTs at 8 seconds and execution times that take longer than this threshold will be left out of the results. These ADTs would take an unreasonably long time to process and are marked with a '-'. Because the results in this table are based on a single execution for every tree, we expect that the results in this table are not close to the true execution time values. Therefore, we cannot draw any further conclusions about the performance from them.

The results for the repetition experiment are shown in the figure below. The figure consists of four graphs with increasing numbers of repetitions. It is important to note that on the x-axis shows the combination of width and height of an ADT. The graphs show the average execution time for the ADT sizes $3 \times 1$ to $3 \times 5$ and $4 \times 1$ to $4 \times 5$ using the conversion tool and the corresponding number of repetitions. In addition to the average execution time, the error bars for each data point in the graph has also been plotted. The area within $\pm 1$ standard deviation for each point has been shaded to visualize the spread of the results for that point.

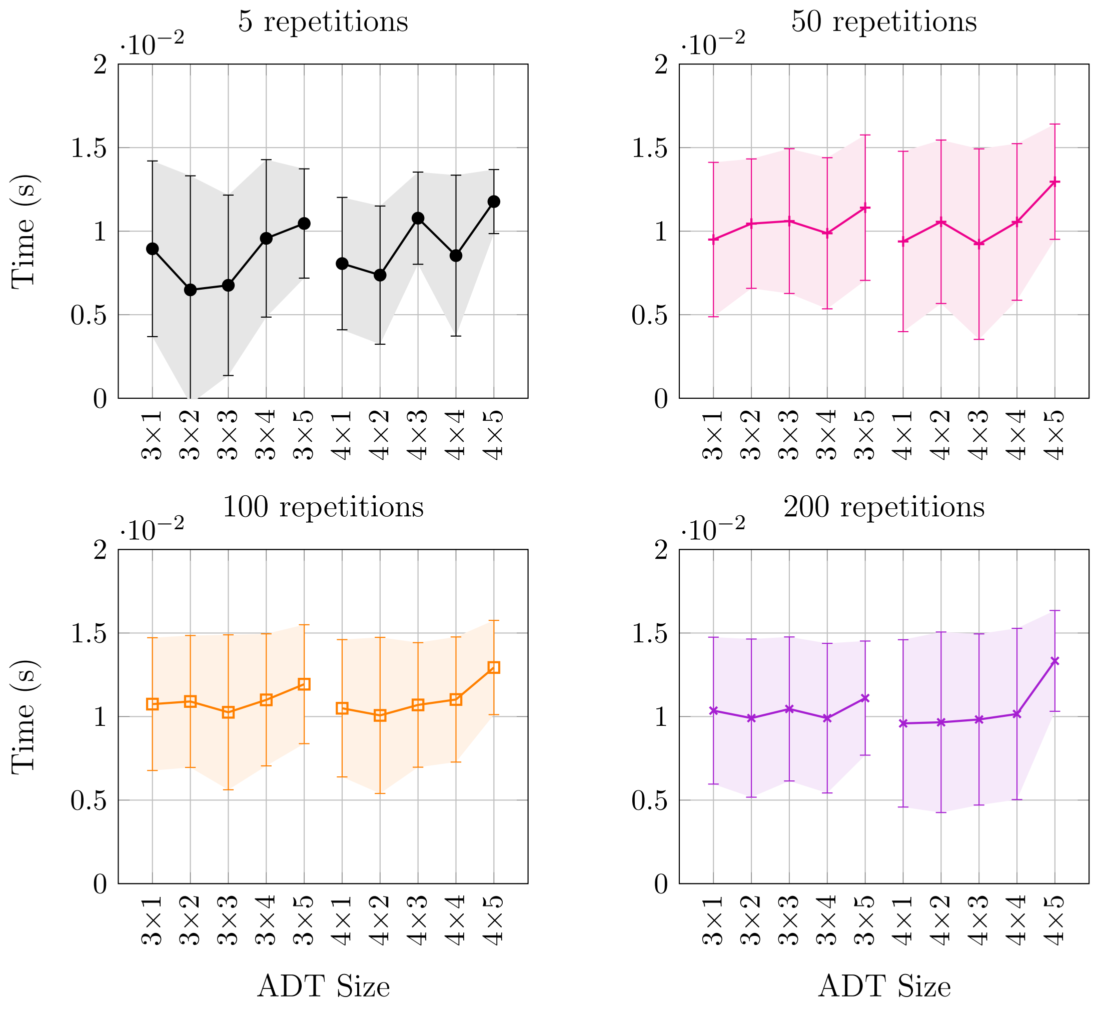

Figure 8: Average execution time for middle size ADTs using the conversion tool and an increasing number of repetitions.

From this figure we can conclude two things, namely, the standard deviation gets smaller for higher numbers of repetitions and the results grow more linearly for higher numbers of repetitions. It is interesting to note that the spread of values seems smaller for 100 repetitions than for 200 repetitions. A reason for this might be that the longer the experiment runs the more uncontrolled effects can influence the process, like background processes or garbage collection being triggered [UR][Mic]. Based on these results we will use 100 repetitions for the experiment that follows.

A sample of the results of the second experiment mentioned in Subsection 4.4.2 are given in this subsection. The complete results of the experiment will be available in appendix B. The results for the conversion tool are shown in Table 6, and the results for the Python module are shown in Table 8. The execution time for smaller trees is too small for the datetime module to notice a difference in time. To ensure we do not get an execution time of 0.0 seconds, we let the process sleep for 1 second and subtract 1 second from the execution time afterwards. We do this for every time we convert an ADT to keep it consistent.

The following graphs show the average execution time to convert the ADTs with both algorithms. For each graph, the x-axis shows the ADT size (width × height) and the y-axis shows the conversion time in seconds. For the first graphs, the ADTs grow in height for each point in the graph and for the last graphs they grow in width. Each graph contains two lines: the cyan line depicts the speed of the conversion tool and the green line depicts the speed of the Python module.

The graphs were separated to improve visualization of processing speeds for all different sizes. The first graph contains narrow ADTs up to a width of 2 and a height of 9. In addition to the average execution time, we have also plotted the error bars and shaded the area within ±1 standard deviation.

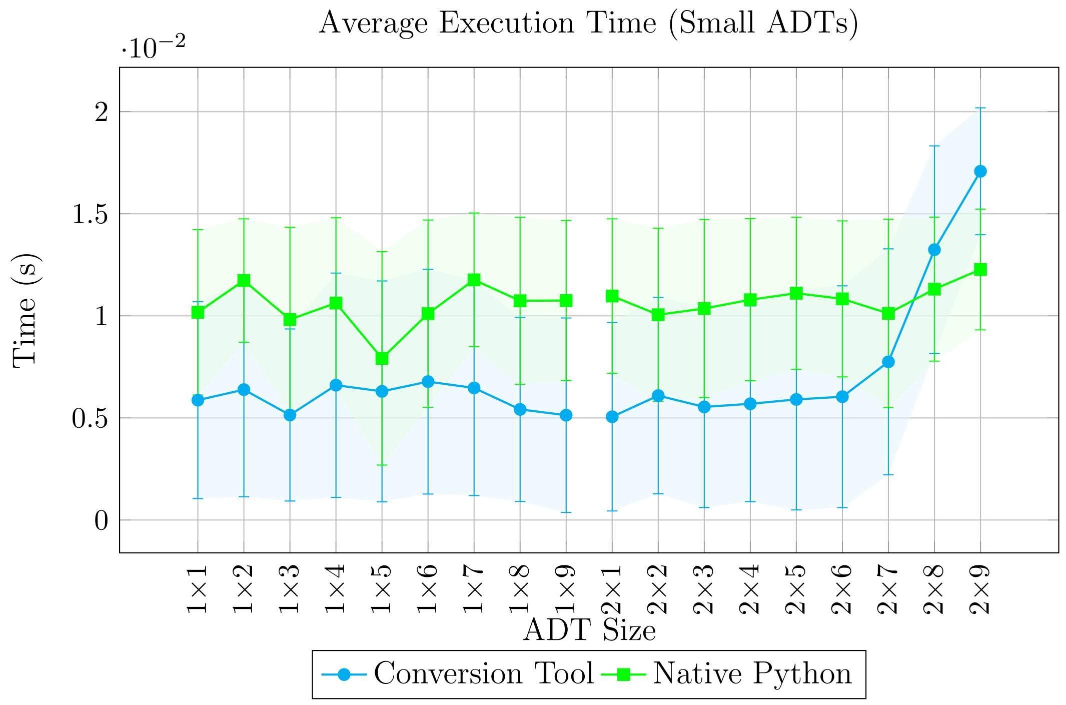

Figure 9: Average execution time for narrow ADT sizes (1×1 to 2×9) on a linear scale.

For these narrow ADTs we can see in Figure 9 that the conversion tool has a smaller average execution time than the Python module. Only when the ADTs get more complex with a combination of a larger width and height the Python module appears to outperform the conversion tool.

The graphs with larger widths contain both small and large trees. The average execution times for these different trees can be far apart, therefore we will use a logarithmic scale to visualize all data points within the same graph. The graph below shows ADT size up to a width of 5 and height of 9.

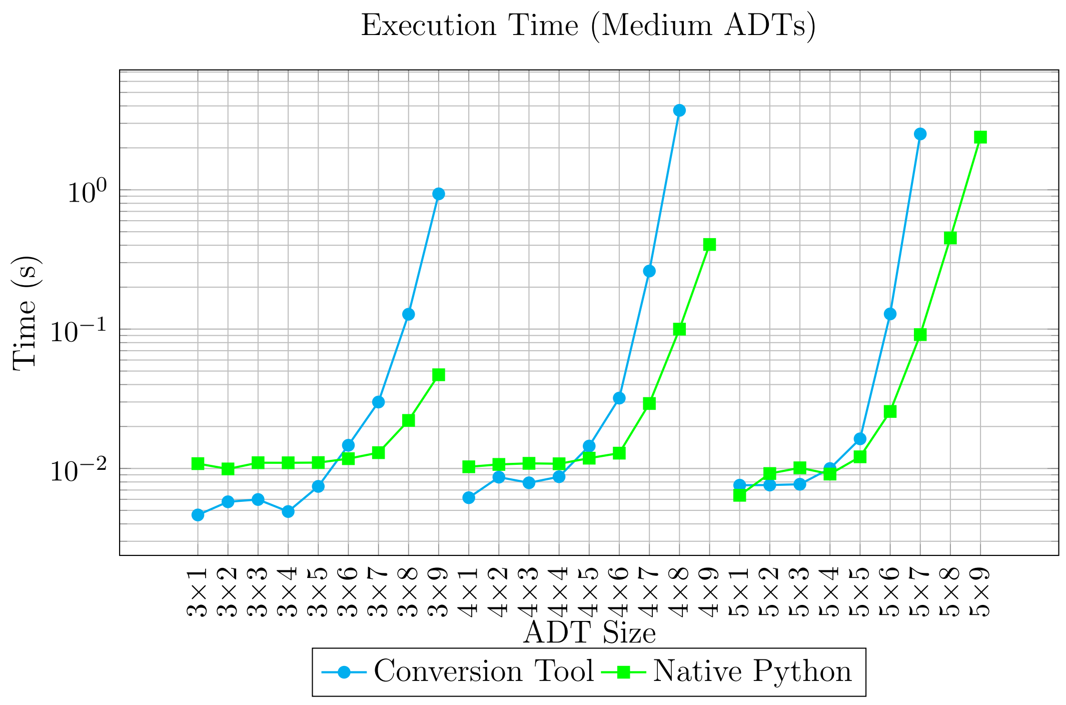

Figure 10: Execution Time for medium large ADT Sizes ($3 \times 1$ to $5 \times 9$) on a logarithmic scale.

From these plots we can see that the difference in speed increases significantly when the ADTs get larger. At a size of 5×7 the Python module has a roughly 27× faster average execution time than the conversion tool, as can be found in Appendix B. For smaller ADTs the conversion tool stays faster in terms of execution time. For an ADT of size $3 \times 4$, the average execution time of the Python module is roughly 2× larger than the conversion tool. The missing values for the execution of larger ADTs using the conversion tool show that the Python module is capable of processing much larger trees in reasonable time than the conversion tool. However, Figure 10 also shows us that for ADTs below the height of 4 the conversion tool keeps having a lower average execution time.

The execution time for larger ADTs with the conversion tool is significantly higher than for the Python module. Yet how extreme this difference really is is not clearly visualized, due to the graphs using a logarithmic scale. To illustrate this, we can plot a graph with a linear scale and only showing larger ADT sizes. The following graph shows ADT sizes from a minimum width of 6 and a minimum height of 1 to a maximum width of 8 and a maximum height of 6.

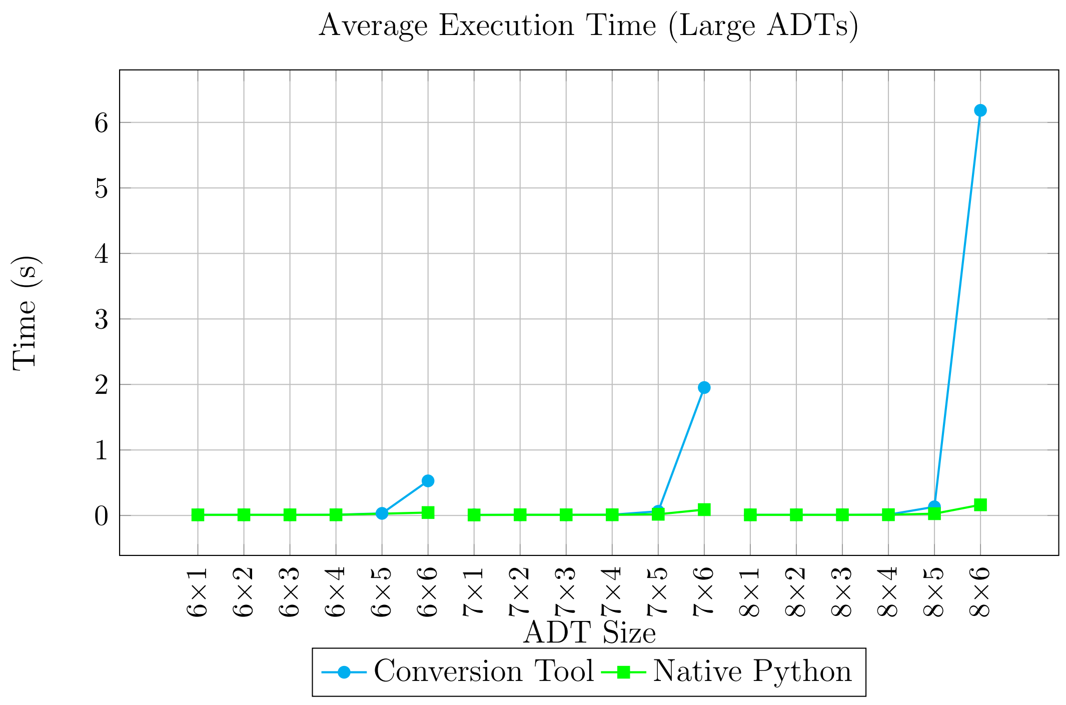

Figure 11: Average execution time for large ADT sizes (6 × 1 to 8 × 7) on a linear scale.

This figure shows that the Python module is fairly quick in processing quite large ADTs. The conversion tool starts struggling with the largest trees and does not show output with these large ADTs, because the execution time exceeds the threshold when the height is more than 6. In the full results we can see that the Python module is able to process even larger trees than the trees that we show in this graph, while the execution time of the conversion tool does not stay within the threshold.

In the following two tables we will increase the width instead of the height. The labels on the x-axis still have the same meaning. For the first graph, we will use a linear scale, because the data points are relatively close to one another.

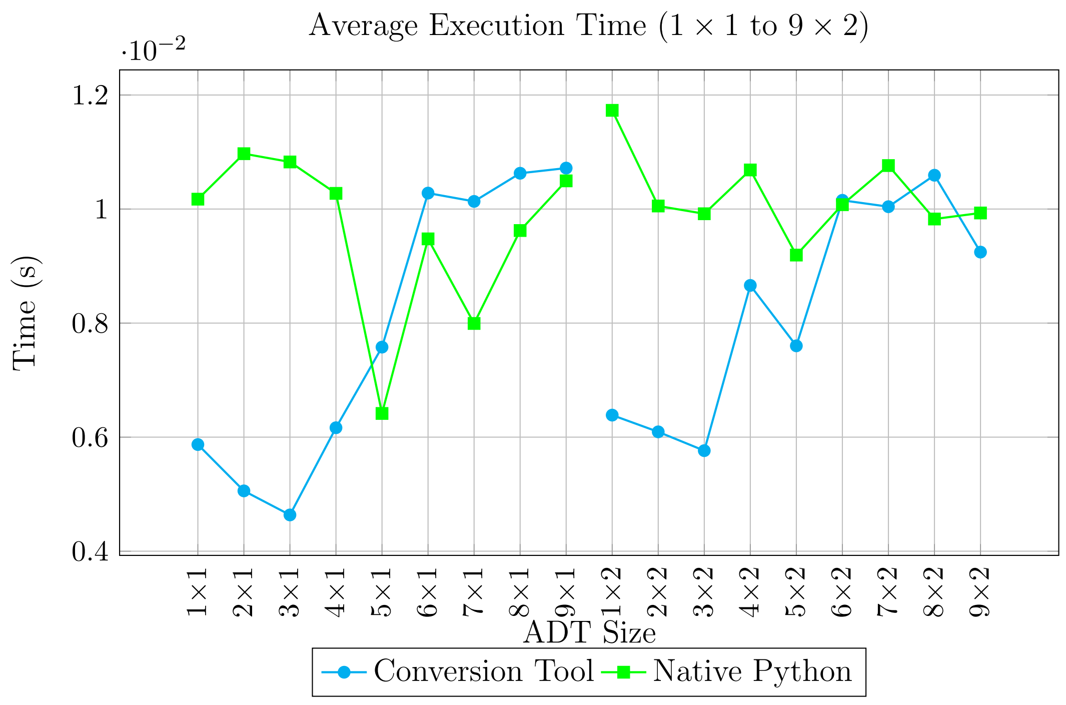

Figure 12: Average execution time for ADT sizes growing in width instead of height (1 × 1 to 9 × 2).

For the first two plots for both tools we keep comparing the same ADTs. This is because for a tree of height 1 the root node has no children, and thus the width has no influence on the size of the tree. This results in an ADT that contains only one node, the root node. Even though this is the same tree, both implementations show inconsistent execution times for these single-node trees. For the average execution times using the conversion tool there appears to be an increase. This is remarkable as the complexity of the trees does not increase. Although the trees of height 2 are increasing in size their average execution time using the Python module stays fairly consistent. For the conversion tool there seems to be a increase in average execution time. These small ADTs have an execution time that is very small, therefore these executions can be more vulnerable to influences from other processes on the system. A small interference might create a relatively big change in the resulting execution time. Especially for the trees of height 1 this can explain the perceived randomness in the results.

To draw conclusions about increasing the width instead of height we need to plot larger trees. Their executions should be less susceptible to outside interferences, as noise is drowned out by their larger execution time. For the following plots we will use trees of height 5, 6, and 7 and increment the width by 1. The graph uses a logarithmic scale, as the results differ substantially.

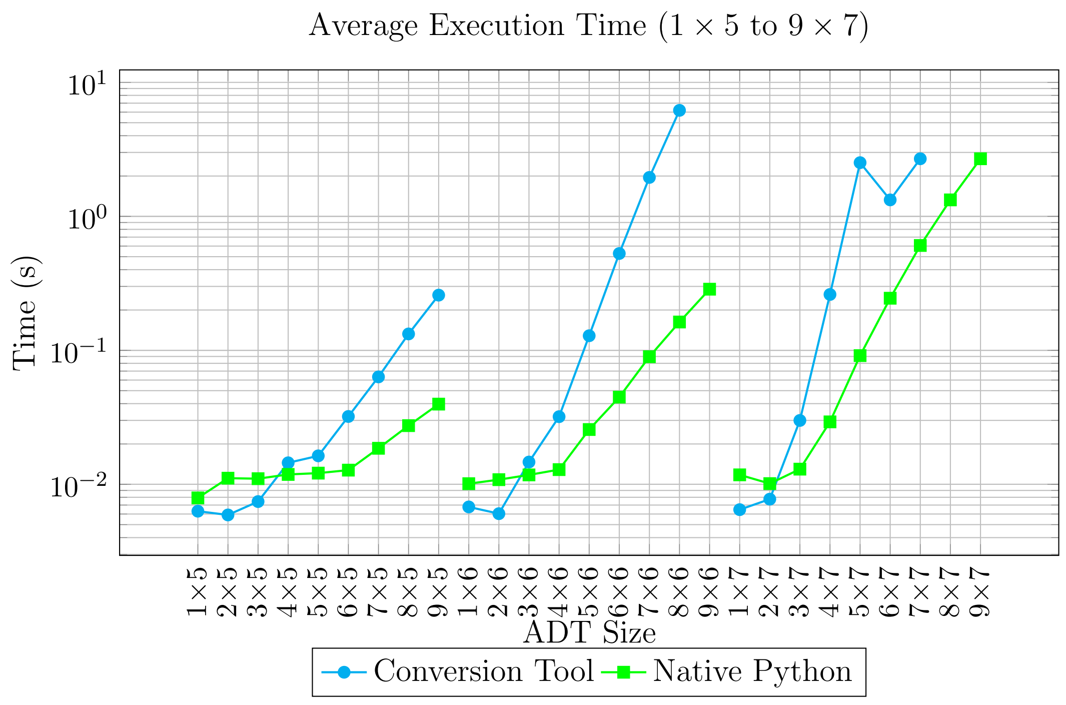

Figure 13: Average execution time for ADT sizes growing in width instead of height ($1 \times 5$ to $9 \times 7$) on a logarithmic scale.

Compared to the plots in Figure 12, the plots in Figure 13 show more consistent growth. When we compare the same plots to 10 we can see that the growth in width causes a less steep growth in average execution time than growth in height. This is especially true for the executions using the Python module. We can see from the file sizes in XML in Table 4 that incrementing the width causes a smaller growth in file size than incrementing the height. For example, the size of $4 \times 7$ is 618974 bytes, the size of $4 \times 8$ is 2650590 bytes and $5 \times 7$ is 2226590 bytes. Processing more bytes costs more time and thus growth in height increases the execution time more than growth in width. Which in turn causes a steeper line in the graph.

# 7 Discussion

In this section we aim to answer the research questions of this thesis, based on the results found in Section 6 and the definition in Subsection 4.3.2.

## 7.1 Definition of an ADT in JSON

In order to ensure compatibility with the XML format established in [BKS13], we have used the XML tags in their format to create JSON objects. However, our implementation of the JSON format has introduced several additions. A part of the additions makes the format more compatible with the overarching project. Another part of the additions is the consistency of JSON, which makes the format more traversable. Some properties of XML attributes, like label, refinement and role, are preserved by making use of key/value pairs. Other properties, like parameters and children, are treated as separate objects, to create a hierarchical relation within the ADT. For children, this is further improved by a naming convention that directly conveys the order of the nodes, and thus ensures the ADT is traversable and adjustable. The root is treated as a node for the same reason. With the predictability and compatibility of our JSON format we will try to reduce the complexity of using ADTs within the overarching project.

## 7.2 Evaluating the Formats

Considering the differences discussed in Subsection 6.1, JSON appears to be the more suitable for our particular problem. Our problem requires that the ADT format can be quickly created and easily updated. These tasks are performed by the computer, and therefore we want to select the format that is most accessible for computers. Due to the structure of the format, JSON is the preferred choice in this context. The file size for JSON is smaller than XML for every file size in the experiment. Both formats have grown roughly at the same rate. However, in Figure 7 we found that JSON file sizes have a higher growth rate for larger ADTs. What might happen eventually is that JSON will overtake XML in size, but this will only happen with much larger ADT sizes. We were not able to find the exact ADT size, because generating such a large ADT in reasonable time is not feasible for our setup. Because ADTs are usually not that large, this concern can be considered negligible. In Subsection 2.1 we saw that ADTs are used for describing security sytems in a formal and methodical manner. In addition, an ADT should be simple enough to give a graphical overview of the attack it is modeled after [MO06]. Therefore, we can argue that an ADT of a size that cannot be visualized is not useful for the purpose it was designed for. On top of that, a computer has to process the bytes of a file. A logical conclusion would be that fewer bytes mean shorter processing time. However, this is not guaranteed and is dependent on more variables than just the file size [FSD]. The format we use is not dependent on the readability of the format for humans, because the visualization of the ADT for humans is done by the overarching project. Therefore, although XML can be argue to be more understandable for humans, we can conclude that JSON is a better choice for handling ADTs than XML, particularly for our use case within the project.

## 7.3 Evaluating the Conversion Algorithm

In Subsection 6.2 we have seen that both algorithms discussed in Subsection 4.2 and Section 5 have advantages in performance. For the conversion tool from Section 5 these are narrow ADTs. Narrow refers to the width of the ADT, or rather the number of children each node has. From the results in Figure 9 and Figure 10 we can see that the conversion tool has consistently lower average execution times for ADTs up to at least a height of 4. ADTs that can be considered large cause the average execution time for both of the algorithms to grow significantly. It is also for these ADTs that the conversion tool loses its advantage over the Python module. The difference in average execution time grows dramatically, as we can see in Figure 11. On top of that, in Figure 13 the growth in average execution time appears to be more stable for the Python module than for the conversion tool. Especially if we combine the results of both Figure 10 and Figure 13. As we have stated in Subsection 7.2 above, we are more interested in the results for smaller, more reasonably sized, ADTs in this project. In these cases the conversion tool clearly outperforms the Python module in terms of average execution time. In addition to that, in Figure 12 we can see a fluctuation in average execution time for single node trees and small trees when using the Python module. This might exist for other tree sizes as well and can lead to inconsistent performance, while our conversion tool seems to keep a more stable average execution time. These observations indicate that our conversion tool is a reasonable candidate for our specific use case.

Though there does exist a JavaScript alternative to our tool, our tool may still be preferred [Nas]. The reason for this is that with our tool we can control the output of our algorithm. The output from xml-js either loses the order of the elements or bloats the output JSON with unnecessary components. Controlling the output means that we can instantly translate both formats to our desired format, while a pre-made library does not take our desired format into account. The result is that we might need to do extra parsing or pruning in order to do modifications.

If the use is scaled to larger ADT sizes, our tool will become obsolete rather quickly. The execution time grows significantly quickly and the tool loses its value as a capable algorithm. With the full results in mind we can see that the Python module is able to process ADTs with a very large number of nodes. Beyond our specific use case we can see that using other conversion solutions may be a better choice.

# 8 Limitations, Conclusion and Future Work

This section will cover the obstacles that shaped our research, summarize our key findings by answering our research questions, and talk about possible further research that can be done in addition to this thesis.

## 8.1 Limitations

During the development of this thesis, we encountered certain difficulties that have limited the scope of our work. These limitations came from design choices, as well as hardware specifications. One notable limitation, for example, is the use of the conversion tool within the overall project. The need to use JavaScript and compatibility with the overall project may have caused limitations both in code performance and structure. The hardware limitations have an influence on the way experiments are run. The ADTs have to be generated and the size of these ADTs depends on the hardware limitations of the computer that is used to generate the ADTs. In this case the limitations are purely concerned with time. The larger the ADTs, the longer it takes to generate them. The hardware might be able to generate bigger ADTs than we used in this thesis, but the time it takes to generate these ADTs would be unreasonably long to be practical. In addition to generating ADTs, the processing of these ADTs also depends on the hardware. If the ADTs become too large conversion processing time may become impractically long and will therefore be cut off by the threshold. The processing time depends not only on the hardware, but also on the algorithm used. This is consistent with the performance limitation stated before. These limitations primarily limit the number of results that can be gathered from the experiments. This can influence the conclusions drawn from the results. Even though this might be the case, we have seen in Section 6 that more results will probably show the same trend. With this in mind we can conclude that the limitation does not significantly affect the results.

## 8.2 Conclusion

For Question 1 of this thesis we have created a way for ADTs to be defined in the JSON file format. We did this by taking the XML format and mimicking its features by either using their JSON counterparts or redefining them into JSON objects. This has led to a JSON definition of ADTs that is both compatible with the XML format and dynamic in terms of modifiability. Question 2 was split up into two sub-questions. The overall goal of the question was achieved by the creation of the conversion tool. In order to evaluate this tool in its usefulness we used the two sub-questions. Question 2.1 addressed the need for such a tool by asking whether or not JSON is a suitable file format for ADTs. We did this by measuring the file sizes of different ADTs in both XML and JSON format and comparing the sizes and growth rate of both. By doing this we could establish that JSON is indeed a viable option to use with ADTs. ADTs of reasonable sizes were consistently smaller in JSON than their XML counterparts. Question 2.2 evaluated the efficiency of our conversion tool by measuring its average execution time for different ADTs and comparing the results to a conversion tool from a Python library. The result showed that our conversion tool had a lower average execution time for most of the smaller ADT sizes. This was enough to argue that our tool had an advantage, because the ADT sizes where the Python library outperformed our tool were unreasonable in size. We also found that our conversion tool was more stable in execution time for the relevant trees.

## 8.3 Future Work

Several aspects of this thesis could be expanded upon or even be changed entirely. Design choices can be altered in order to improve the functionality of the conversion tool and experiments can be added to further analyze the working of the algorithm. In this thesis we wrote our tool in JavaScript, and we have seen from the results that there can be better solutions using different algorithms than the one we came up with. An example would be to try writing a recursive algorithm instead of the iterative one we used for the conversion tool. Using a different language, like Python, or finding better functionalities within JavaScript can make the tool run more efficiently. To use Python or other languages, integration methods need to be found in order to make it fully functional within the overall project. Otherwise, the JavaScript environment would be unable to process the results. An important part of this thesis was comparing the algorithm we made to another relevant implementation. Conducting additional experiments might enable us to draw different conclusions. In the experiments done in this thesis we did not fill ADTs with more information than children and labels. An interesting experiment would be to determine the most efficient way to implement populated ADTs. Experiments can be limited to smaller trees, as visualization in this project is purely meant for human use and large trees are not easy to understand for humans. In this thesis we limited ourselves to converting only between XML and JSON; however, there can be more relevant formats to be converted to and from. The goal of the project around this thesis is to improve the usability, which means a case could be made for supporting additional conversions.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<adtree>
    <node refinement="conjunctive">
        <label>steal money</label>
        <node refinement="disjunctive">
            <label>learn PIN</label>
            <node refinement="disjunctive">
                <label>find PIN</label>
                <parameter domainId="MinTimeSeq1">100.0</parameter>
                <parameter domainId="ProbSucc2">0.1</parameter>
            </node>
        </node>
    </node>
</adtree>
```

XML

```json
{
    "0": {
        "label": "steal money",
        "refinement": "1",
        "switchRole": "0",
        "0":{
            "label": "learn PIN",
            "refinement": "0",
            "switchRole": "0",
            "0":{
                "label": " find PIN",
                "refinement": "0",
                "switchRole": "0",
                "parameters": [
                    "1":{
                        "name": "MinTimeSeq1",
                        "value": 100.0
                    },
                    "2":{
                        "name": "ProbSucc2",
                        "value": 0.1
                    }
                ]
            },
            "parameters": []
        }
    }
}
```

JSON

Figure 14: Illustration of the same data encoded in XML and JSON.

## References

[Agu] C. Aguilar. Simple xml2json parser. Œhttps://github.com/buglabs/node-xml2json. Accessed: 03-12-2025.

[AIJ] Adts in json. Œhttps://github.com/stevenvp05/ADTs-in-JSON.

[ASY] Asynchronous javascript. ŒŒhttps://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous. Accessed: 30-04-2024.

[BKS11] S. Radomirović B. Kordy, S. Mauw and P. Schweitzer. Foundations of attack–defense trees. *Lecture Notes in Computer Science*, 2011.

[BKS13] S. Mauw B. Kordy, P. Kordy and P. Schweitzer. Adtool: Security analysis with attack-defense trees extended version. 2013.

[Ble] M. Blech. xmltodict: Makes working with xml feel like you are working with json. Œhttps://pypi.org/project/xmltodict/. Accessed: 25-06-2024.

[B17] D. Tošić B. Šandrih and V Filipović. Towards efficient and unified xml/json conversion - a new conversion. *IPSI BgD Transactions on Internet Research (TIR) vol 13*, 2017.

[B17] Z. Wilimowska B. Świecicki, L. Borzemski and J. Światek. *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems*. Springer International Publishing AG, 2017. Accessed: 19-10-2025.

[Car18] P. A. Carter. *QL Server Advanced Data Types*. Apress Berkeley, 2018. Accessed: 19-10-2025.

[Col15] T. J. Cole. Too many digits: The presentation of numerical data. 2015. Accessed: 02-12-2025.

[Coo] Clark Cooper. Using expat. Œhttps://www.xml.com/pub/1999/09/expat/index.html. Accessed: 26-06-2024.

[DB] P. Wendler D. Beyer, S. Löwe. Reliable benchmarking: requirements and solutions. Œhttps://www.sosy-lab.org/research/pub/2019-STTT.Reliable_Benchmarking_Requirements_and_Solutions.pdf. Accessed: 07-11-2025.

[DOMa] Document object model (dom). https://www.ionos.com/digitalguide/websites/web-development/an-introduction-to-the-document-object-model-dom/. Accessed: 19-02-2024.

[DOMb] Domparser. Œhttps://developer.mozilla.org/en-US/docs/Web/API/DOMParser. Accessed: 28-06-2024.

[DT] datetime — basic date and time types. Œhttps://docs.python.org/3/library/datetime.html. Accessed: 2-10-2024.

[DVDdW08] G. Martens E.Mannens D. Van Deursen, C. Poppe and R. Van de Walle. Xml to rdf conversion: a generic approach. *2008 International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution*, 2008.

[FSA] File system api. `Œhttps://developer.mozilla.org/en-US/docs/Web/API/File_System_API`. Accessed: 10-12-2025.

[FSD] File size distribution on unix systems—then and now. `Œhttps://www.cs.vu.nl/~herbertb/papers/filesize_osr2006.pdf`. Accessed: 14-07-2025.

[GLO] Standard built-in objects. `Œhttps://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects`. Accessed: 20-10-2025.

[GWM17] S. Yang G. Werner and K. McConky. Time series forecasting of cyber attack intensity. *Proceedings of the 12th Annual Conference on Cyber and Information Security Research - CISRC*, 2017.

[HRD] Human readability of data files. `Œhttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC11034916/`. Accessed: 21-07-2024.

[JED] json — json encoder and decoder. `Œhttps://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON`. Accessed: 27-06-2024.

[JJM] Json. `Œhttps://docs.python.org/3/library/json.html`. Accessed: 27-06-2024.

[JSO] Introducing json. `Œhttps://www.json.org/json-en.html`. Accessed: 02-04-2024.

[Kha15] I. A. Khan. Xml and json translator. 2015. Accessed: 07-11-2025.

[KJMT18] C. M. Jakobson K. J. Metcalf, M. F. Slininger Lee and D. Tullman-Ercek. An estimate is worth about a thousand experiments: using order-of-magnitude estimates to identify cellular engineering targets. 2018. Accessed: 01-12-2025.

[LiL] A list of open-source c++ libraries. `Œhttps://en.cppreference.com/w/cpp/links/libs`. Accessed: 27-06-2024.

[Mic] Microsoft. Garbage collection and performance. `Œhttps://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/performance`. Accessed: 05-12-2025.

[MO06] S. Mauw and M. Oostdijk. Foundations of attack trees. *Information Security and Cryptology - ICISC 2005*, 2006.

[Nas] Y. Nashwaan. xml-js. `Œhttps://github.com/nashwaan/xml-js`. Accessed: 03-12-2025.

[NNI09] R. Reynolds N. Nurseitov, M. Paulson and C. Izurieta. Comparison of json and xml data interchange formats: A case study. 2009.

[Nyb14] M. A. Nyblom. On the average path length of complete m-ary trees. 2014. Accessed: 10-12-2025.

[OGTR16] P. Kordy K. Lounis S. Mauw O. Gadyatskaya, R. Jhawar and R. Trujillo-Rasua. Attack trees for practical security assessment: Ranking of attack scenarios with adtool 2.0. *Quantitative Evaluation of Systems. Lecture Notes in Computer Science*, 2016.

[PSX] Parsing and serializing xml. Œ`https://developer.mozilla.org/en-US/docs/Web/XML/Parsing_and_serializing_XML`. Accessed: 26-06-2024.

[RFN] Reading files with node.js. Œ`https://nodejs.org/en/learn/manipulating-files/reading-files-with-nodejs`. Accessed: 10-12-2025.

[RJRR15] S. Mauw S. Radomirović R. Jhawar, B. Kordy and R.Trujillo-Rasua. Attack trees with sequential conjunction. 2015. Accessed: 18-12-2025.

[Sch99] B. Schneier. Attack trees : Modeling security threats. *Dr. Dobb's Journal*, 1999.

[Sel15] K. Seldor. The law of large numbers and its applications. 2015. Accessed: 07-11-2025.

[SG25] N. D. Schiele and O. Gadyatskay. Attack tree distance: a practical examination of tree difference measurement within cyber security. 2025. Accessed: 18-12-2025.

[ŠM14] A. Šimec and A. Magličić. Comparison of json and xml data formats. 2014. Accessed: 27-10-2025.

[SOP] Simple object access protocol (soap) 1.1. Œ`www.w3.org/TR/2000/NOTE-SOAP-20000508/`. Accessed: 15-07-2025.

[TLH18a] P. Yan T. Lv and W. He. Survey on json data modelling. *Journal of Physics: Conference Series*, 2018.

[TLH18b] Ping Yan Teng Lv and Weimin He. Survey on json data modelling. *Journal of Physics: Conference Series*, 2018. Accessed: 01-05-2024.

[TWL22] F. Delcourt T. Weiser and A. Legay. Graphical user interface for dag-based attack trees. *"http://hdl.handle.net/2078.1/thesis:35682"*, 2022.

[UR] D. Uroz and R. J. Rodríguez. Characteristics and detectability of windows auto-start extensibility points in memory forensics. Œ`https://webdiis.unizar.es/~ricardo/files/papers/UR-DIIN-19.pdf`. Accessed: 05-12-2025.

[WAD] Web api design crafting interfaces that developers love. Œ`https://www.accorsi.net/docs/api-design-ebook-2012-03.pdf`. Accessed: 15-07-2025.

[WWP14] B. Fila W. Wideł, M. Audinot and S. Pinchinat. Formal methods for attack tree-based security modeling. 2014. Accessed: 18-12-2025.

[XMLa] Extensible markup language (xml). Œ`https://www.w3.org/XML/`. Accessed: 12-04-2024.

[XMLb] Extensible markup language (xml) 1.0 (fifth edition). Œ`https://www.w3.org/TR/REC-xml/`. Accessed: 27-05-2024.

[XPM] Xml processing modules. Œ`https://docs.python.org/3/library/xml.html`. Accessed: 27-06-2024.

[ZD14] S. Zunke and V. D'Souza. Json vs xml: A comparative performance analysis of data exchange formats. *IJCSN International Journal of Computer Science and Network*, 3(4):257–261, 2014.

[Zha07] Q. Zhang. Embedding parallel bit stream technology into expat. 2007. Accessed: 03-12-2025.

[ZUHH15] G. F. Khan Z. U. Haq and T. Hussain. A comprehensive analysis of xml and json web technologies. 2015.

# 9 Appendix

## A Data Format Examples

The following listing is an example of an JSON ADT.

```
{
    "adtree": {
        "label": "Bank Account",
        "refinement": "disjunctive",
        "switchRole": "no",
        "children": {
            {
                "label": "ATM",
                "refinement": "conjunctive",
                "switchRole": "no",
                "children": {
                    {
                        "label": "Card",
                        "refinement": "disjunctive",
                        "switchRole": "no",
                        "children": {},
                        "parameters": {
                            {
                                "name": "MinTimeSeq1",
                                "value": 20.0
                            },
                            {
                                "name": "ProbSucc2",
                                "value": 0.25
                            }
                        }
                    }
                    {
                        "label": "PIN",
                        "refinement": "disjunctive",
                        "switchRole": "no",
                        "children": {
                            {
                                "label": "Eavesdrop",
                                "refinement": "disjunctive",
                                "switchRole": "no",
                                "children": {},
                                "parameters": {
                                    {
                                        "name": "MinTimeSeq1",
                                        "value": 100.0
                                    },
                                    {
                                        "name": "ProbSucc2",
                                        "value": 0.5
                                    }
                                }
                            },
                            {
                                "label": "Find Note",
                                "refinement": "disjunctive",
                                "switchRole": "no",
                                "children": {
                                    {
                                        "label": "Memorize",
                                        "refinement": "disjunctive",
                                        "switchRole": "yes",
                                        "children": {
                                            {
                                                "label": "Force",
                                                "refinement": "disjunctive",
                                                "switchRole": "no",
                                                "children": {},
                                                "parameters": {
                                                    {
                                                        "name": "MinTimeSeq1",
                                                        "value": 5.0
                                                    },
                                                    {
                                                        "name": "ProbSucc2",
                                                        "value": 0.9
                                                    }
                                                }
                                            }
                                        },
                                        "parameters": {}
                                    }
                                },
                                "parameters": {
                                    {
                                        "name": "MinTimeSeq1",
                                        "value": 100.0
                                    },
                                    {
                                        "name": "ProbSucc2",
                                        "value": 0.1
                                    }
                                }
                            }
                        },
                        "parameters": {}
                    }

                },
                "parameters": {}
            }
            {
                "label": "Online",
                "refinement": "conjunctive",
                "switchRole": "no",
                "children": {
                    {
                        "label": "Password",
                        "refinement": "disjunctive",
                        "switchRole": "no",
                        "children": {
                            {
                                "label": "Phishing",
                                "refinement": "disjunctive",
                                "switchRole": "no",
                                "children": {},
                                "parameters": {}
                            },
                            {
                                "label": "Key Logger",
                                "refinement": "disjunctive",
                                "switchRole": "no",
                                "children": {},
                                "parameters": {}
                            },
                            {
                                "label": "2nd Auth Factor",
                                "refinement": "disjunctive",
                                "switchRole": "yes",
                                "children": {
                                    {
                                        "label": "Key Fobs",
                                        "refinement": "disjunctive",
                                        "switchRole": "yes",
                                        "children": {},
                                        "parameters": {}
                                    },
                                    {
                                        "label": "PIN Pad",
                                        "refinement": "disjunctive",
                                        "switchRole": "yes",
                                        "children": {},
                                        "parameters": {}
                                    },
                                    {
                                        "label": "Malware",
                                        "refinement": "disjunctive",
                                        "switchRole": "no",
                                        "children": {
                                            {
                                                "label": "Browser",
                                                "refinement": "disjunctive",
                                                "switchRole": "no",
                                                "children": {},
                                                "parameters": {}
                                            },
                                            {
                                                "label": "OS",
                                                "refinement": "disjunctive",
                                                "switchRole": "no",
                                                "children": {},
                                                "parameters": {}
                                            }
                                        },
                                        "parameters": {}
                                    }
                                },
                                "parameters": {}
                            }
                        },
                        "parameters": {}
                    },
                    {
                        "label": "Username",
                        "refinement": "disjunctive",
                        "switchRole": "no",
                        "children": {},
                        "parameters": {}
                    }
                },
                "parameters": {}
            }
        }
    }
}
```

Listing 3: Full Example JSON ADT

The listing below shows the complete XML example from figure 13 [BKS13].

```
<?xml version="1.0" encoding="UTF-8"?>
<adtree>
    <node refinement="conjunctive">
        <label>steal money</label>
        <node refinement="disjunctive">
            <label>learn PIN</label>
            <node refinement="disjunctive">
                <label>eavesdrop PIN</label>
                <parameter domainId="MinTimeSeq1">10.0</parameter>
                <parameter domainId="ProbSucc2">0.5</parameter>
            </node>
            <node refinement="disjunctive">
                <label>find PIN</label>
                <parameter domainId="MinTimeSeq1">100.0</parameter>
                <parameter domainId="ProbSucc2">0.1</parameter>
            </node>
            <node refinement="conjunctive" switchRole="yes">
                <label>security training</label>
                <node refinement="disjunctive">
                    <label>learn rules</label>
                    <parameter domainId="ProbSucc2">0.7</parameter>
                </node>
                <node refinement="disjunctive">
                    <label>adhere to rules</label>
                    <parameter domainId="ProbSucc2">0.3</parameter>
                    <node refinement="disjunctive" switchRole="yes">
                        <label>threaten victim</label> 28 <parameter domainId="MinTimeSeq1">5.0</parameter>
                        <parameter domainId="ProbSucc2">0.9</parameter>
                    </node>
                </node>
            </node>
        </node>
        <node refinement="disjunctive">
            <label>get card</label>
            <parameter domainId="MinTimeSeq1">20.0</parameter>
            <parameter domainId="ProbSucc2">0.25</parameter>
        </node>
    </node>
    <domain id="MinTimeSeq1">
        <class>lu.uni.adtool.domains.predefined.MinTimeSeq</class>
        <tool>ADTool</tool>
    </domain>
    <domain id="ProbSucc2">
        <class>lu.uni.adtool.domains.predefined.ProbSucc</class>
        <tool>ADTool</tool>
    </domain>
</adtree>
```

Listing 4: ADTool figure 13

## B Results of the Experiments

### B.1 Results for Data Format Comparison

The following tables consist of the results for the file size comparisons.

| ADT Size | XML | JSON |
|---|---|---|
| 1x1 | 75 | 38 |
| 2x2 | 187 | 97 |
| 3x3 | 765 | 412 |
| 4x4 | 5067 | 2825 |
| 5x5 | 48055 | 27582 |
| 6x6 | 593475 | 349303 |
| 7x7 | 9013233 | 5424932 |
| 8x8 | 162293899 | 99678925 |
| 9x9 | 3377822403 | 2113324966 |

Table 2: File sizes for varying ADT sizes incrementing by 1 width and 1 height per row.

| ADT Size | XML | JSON |
|---|---|---|
| 1x1 | 75 | 38 |
| 2x1 | 131 | 67 |
| 2x2 | 187 | 97 |
| 2x3 | 243 | 127 |
| 2x4 | 299 | 157 |
| 2x5 | 355 | 187 |
| 3x1 | 189 | 98 |
| 3x2 | 419 | 223 |
| 3x3 | 765 | 412 |
| 3x4 | 1227 | 665 |
| 3x5 | 1805 | 982 |
| 4x1 | 249 | 131 |
| 4x2 | 899 | 491 |
| 4x3 | 2385 | 1321 |
| 4x4 | 5067 | 2825 |
| 4x5 | 9305 | 5207 |
| 5x1 | 311 | 166 |
| 5x2 | 1891 | 1059 |
| 5x3 | 7407 | 4210 |
| 5x4 | 20939 | 11977 |
| 5x5 | 48055 | 27582 |

Table 3: File sizes for varying ADT sizes incrementing by 1 height up to an height of 5 per row for widths of 1, 2, 3, 4 and 5

| | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Height | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 110 | 110 | 110 | 110 | 110 | 110 | 110 | 110 | 110 |
| 2 | 186 | 262 | 338 | 414 | 490 | 566 | 642 | 718 | 794 |
| 3 | 270 | 598 | 1094 | 1758 | 2590 | 3590 | 4758 | 6094 | 7598 |
| 4 | 362 | 1334 | 3578 | 7646 | 14090 | 23462 | 36314 | 53198 | 74666 |
| 5 | 462 | 2934 | 11678 | 33246 | 76590 | 153062 | 276414 | 462798 | 730766 |
| 6 | 570 | 6390 | 37922 | 143838 | 414090 | 992870 | 2091570 | 4001742 | 7108058 |
| 7 | 686 | 13814 | 122486 | 618974 | 2226590 | 6404966 | 15738854 | 34410446 | 68755214 |
| 8 | 810 | 29686 | 393674 | 2650590 | 11914090 | 41117030 | 117858186 | 294457294 | 661843370 |
| 9 | 942 | 63478 | 1259726 | 11301342 | 63476590 | 262826342 | 878811918 | 2509049806 | 6344010542 |

Table 4: File sizes for varying ADT sizes in XML.

| Size step | XML | JSON |
|---|---|---|
| 1x1 to 2x2 | 2.49 | 2.55 |
| 2x2 to 3x3 | 4.09 | 4.25 |
| 3x3 to 4x4 | 6.62 | 6.86 |
| 4x4 to 5x5 | 9.49 | 9.77 |
| 5x5 to 6x6 | 12.35 | 12.66 |
| 6x6 to 7x7 | 15.18 | 15.53 |
| 7x7 to 8x8 | 18.01 | 18.38 |
| 8x8 to 9x9 | 20.82 | 21.21 |

Table 5: File size ratios between varying ADT sizes.

### B.2 Results for Algorithm Evaluation

For the following tables the numbers above each column correspond with the width of each tree. Numbers next to each row correspond to the height of the tree. Time is measured in seconds. Results are cut off at a threshold of 8 seconds and missing results are marked by a '-'. The results have been rounded to four decimals to ensure that the standard deviation of every result has at least two significant digits [Col15].

| | | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Height | 1 | 0.0059 | 0.0051 | 0.0046 | 0.0062 | 0.0076 | 0.0103 | 0.0101 | 0.0106 | 0.0107 |
| | 2 | 0.0064 | 0.0061 | 0.0058 | 0.0087 | 0.0076 | 0.0102 | 0.0100 | 0.0106 | 0.0093 |
| | 3 | 0.0051 | 0.0055 | 0.0060 | 0.0079 | 0.0077 | 0.0098 | 0.0099 | 0.0098 | 0.0104 |
| | 4 | 0.0066 | 0.0057 | 0.0049 | 0.0087 | 0.0100 | 0.0116 | 0.0119 | 0.0126 | 0.0140 |
| | 5 | 0.0063 | 0.00591 | 0.0074 | 0.0145 | 0.0163 | 0.0321 | 0.0634 | 0.1326 | 0.2582 |
| | 6 | 0.0068 | 0.0060 | 0.0147 | 0.0320 | 0.1285 | 0.5283 | 1.9529 | 6.1841 | – |
| | 7 | 0.0065 | 0.0078 | 0.0300 | 0.2611 | 2.5167 | – | – | – | – |
| | 8 | 0.0054 | 0.0132 | 0.1278 | 3.7176 | – | – | – | – | – |
| | 9 | 0.0051 | 0.0179 | 0.9349 | – | – | – | – | – | – |

Table 6: Average Execution Time for varying ADT sizes using the conversion tool.

| | | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| height | 1 | 0.0048 | 0.0046 | 0.0040 | 0.0054 | 0.0058 | 0.0041 | 0.0039 | 0.0036 | 0.0038 |
| | 2 | 0.0053 | 0.0048 | 0.0052 | 0.0059 | 0.0058 | 0.0036 | 0.0043 | 0.0040 | 0.0045 |
| | 3 | 0.0042 | 0.0049 | 0.0049 | 0.0057 | 0.0059 | 0.0044 | 0.0045 | 0.0046 | 0.0033 |
| | 4 | 0.0055 | 0.0048 | 0.0042 | 0.0055 | 0.0059 | 0.0031 | 0.0034 | 0.0035 | 0.0041 |
| | 5 | 0.0054 | 0.0054 | 0.0056 | 0.0042 | 0.0032 | 0.0058 | 0.0050 | 0.0083 | 0.0062 |
| | 6 | 0.0055 | 0.0054 | 0.0046 | 0.0033 | 0.0047 | 0.0079 | 0.0290 | 0.1016 | – |
| | 7 | 0.0053 | 0.0055 | 0.0045 | 0.0115 | 0.0379 | – | – | – | – |
| | 8 | 0.0045 | 0.0051 | 0.0044 | 0.1185 | – | – | – | – | – |
| | 9 | 0.0048 | 0.0031 | 0.0112 | – | – | – | – | – | – |

Table 7: Standard deviation corresponding to varying ADT sizes using the conversion tool.

| | | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Height | 1 | 0.0102 | 0.0110 | 0.0108 | 0.0103 | 0.0064 | 0.0095 | 0.0080 | 0.0096 | 0.0105 |
| | 2 | 0.0117 | 0.0101 | 0.0099 | 0.0107 | 0.0092 | 0.0101 | 0.0108 | 0.0098 | 0.0099 |
| | 3 | 0.0098 | 0.0104 | 0.0110 | 0.0109 | 0.0101 | 0.0099 | 0.0103 | 0.0102 | 0.0094 |
| | 4 | 0.0106 | 0.0108 | 0.0110 | 0.0108 | 0.0091 | 0.0101 | 0.0107 | 0.0117 | 0.0118 |
| | 5 | 0.0079 | 0.0111 | 0.0110 | 0.0119 | 0.0121 | 0.0128 | 0.0186 | 0.0274 | 0.040 |
| | 6 | 0.0101 | 0.0108 | 0.0118 | 0.0129 | 0.0256 | 0.0447 | 0.0896 | 0.1627 | 0.2860 |
| | 7 | 0.0118 | 0.0101 | 0.0130 | 0.0293 | 0.0913 | 0.2448 | 0.6063 | 1.3272 | 2.6914 |
| | 8 | 0.0107 | 0.0113 | 0.0221 | 0.1000 | 0.4512 | 1.5680 | 4.4590 | – | – |
| | 9 | 0.0108 | 0.0123 | 0.0470 | 0.4050 | 2.3882 | – | – | – | – |

Table 8: Results for varying ADT sizes using the Python module.

| | | Width | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Height | 1 | 0.0040 | 0.0038 | 0.0039 | 0.0046 | 0.0050 | 0.0044 | 0.0050 | 0.0045 | 0.0041 |
| | 2 | 0.0030 | 0.0042 | 0.0046 | 0.0045 | 0.0044 | 0.0045 | 0.0040 | 0.0038 | 0.0038 |
| | 3 | 0.0045 | 0.0044 | 0.0038 | 0.0038 | 0.0040 | 0.0043 | 0.0041 | 0.0040 | 0.0044 |
| | 4 | 0.0042 | 0.0040 | 0.0036 | 0.0040 | 0.0043 | 0.0041 | 0.0038 | 0.0034 | 0.0032 |
| | 5 | 0.0052 | 0.0037 | 0.0035 | 0.0031 | 0.0035 | 0.0032 | 0.0053 | 0.0035 | 0.0051 |
| | 6 | 0.0046 | 0.0038 | 0.0032 | 0.0028 | 0.0046 | 0.0039 | 0.0056 | 0.0067 | 0.0066 |
| | 7 | 0.0033 | 0.0046 | 0.0024 | 0.0033 | 0.0069 | 0.0086 | 0.0148 | 0.0433 | – |
| | 8 | 0.0041 | 0.0035 | 0.0058 | 0.0069 | 0.0230 | 0.0714 | 0.0966 | 0.0105 | – |
| | 9 | 0.0039 | 0.0030 | 0.0044 | 0.0172 | 0.0752 | – | – | – | – |

Table 9: Standard deviation corresponding to varying ADT sizes using the Python module.