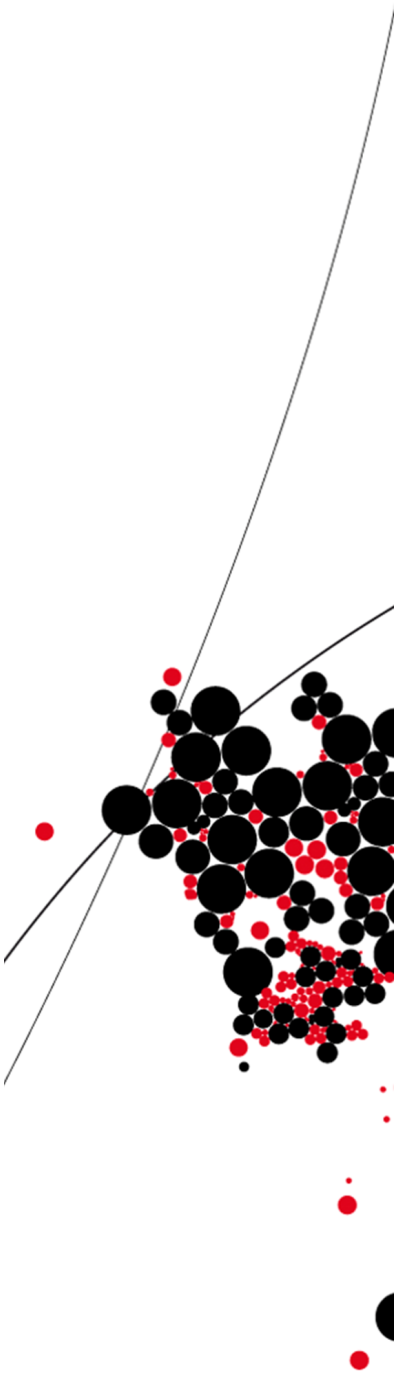
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



Automated Generation of Attack Trees by Unfolding Graph Transformation Systems

David J. Huistra

M.Sc. Thesis
March 2016

Supervisors:
prof. dr. ir. A. Rensink
dr. ir. M.I.A. Stoelinga
D. Ionita, MSc

Formal Methods & Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Analysis of an organization's security and the threats it faces is nowadays often done using attack trees that describe all possible threats facing a system or organization. A big challenge lies in obtaining these attack trees. Manually constructing them is tedious and error-prone work. Therefore, this project focuses on generating attack trees automatically from a given model that describes a system or organization. It improves upon previous efforts by providing an approach to identify all possible attacks from a given model in a more scalable manner, compared to the previous approach of constructing an attack graph, while remaining (security-)domain independent.

This work demonstrates that this new approach, based on partial-order reduction, can have significant scalability benefits compared to the existing generic approach, although this scalability improvement is related to the amount of concurrent actions in the organization's model.

In addition, it is shown that the graph transformations modeling paradigm can be used as a generic input language for describing systems and organizations, and using graph transformations gives the benefit of reusing existing efforts and implementations. Specifically, a partial-order technique called the unfolding of a graph transformation system is used as the basis of the approach, and GROOVE, a tool for constructing and analysing graph transformation systems, is used as the basis of the implementation.

Finally, this work demonstrates that using partial-order reduction to identify all possible attacks also provides the approach with sufficient information to determine if attack steps should be executed in sequence or not, or if two attack steps can both be performed during the same attack or not. This allows the approach to add SQAND and XOR gates to the constructed attack trees, in addition to regular AND and OR gates, providing analysis tools with more analysis opportunities.

Acknowledgements

What a Long, Strange Trip It's Been

Achievement, *World of Warcraft*

This thesis marks the end of my study Computer Science. Through this final report, I would like to acknowledge the people that have made the last six and a half years the best period of my life (so far).

First off all I want to thank my parents for always being supportive and ready to assist and offer advice whenever it was needed (and even when it wasn't)! I also want to thank the rest of my family for all their encouragements, which kept me going when my motivation was low.

I also want to acknowledge all of the people who I have known from the start of my studies and still consider friends: My housemates and do-group members. Thank you for learning me to drink beer! You all have made this period truly memorable through the many adventures we undertook.

The last seven months I have worked overtime on my Master Thesis. While this was a period of hard work, it was also a unique and rewarding experience to discover what the world of research entails. I want thank all of the FMT research group for warmly welcoming me into your group. By allowing me to work in your offices and to join your lunch colloquiums, your stand ups, the Friday afternoon drinks and break-room discussion, I felt part of the team and had an educative and above all wonderful time!

I want to specially thank my supervisors for always being enthusiastic about the research, motivating me and pushing me to keep challenging myself. Thank you for always making the time to have a discussion and to provide feedback from different viewpoints. I can truly say that I have learned a lot more than I anticipated because of this!

I also want to thank all of the students who were also working on their thesis during this time and with whom I could discuss my issues, progress and frustrations. Thank you Vincent, Robin, Kaspar, Bas, Jochem, Niels, Herman and those I am forgetting right now :)

Finally, I had the fortune of meeting a lot of wonderful people during my study time, too many to thank here without forgetting people, so I hereby

want to thank all of you with whom I worked together on a study project, an Inter-*Actief* committee or a project at Topicus. Everybody I had an inspiring conversation with or simply shared a beer with. All of you motivated me to work my hardest on this thesis to put a crown on this amazing period of my life.

— *David Huistra*
March 2016

Contents

Chapter 1

Introduction

> (...) it becomes increasingly clear that the term "security" doesn't have meaning unless also you know things like "Secure from whom?" or "Secure for how long?"
>
> Clearly, what we need is a way to model threats against computer systems. If we can understand all the different ways in which a system can be attacked, we can likely design countermeasures to thwart those attacks. And if we can understand who the attackers are – not to mention their abilities, motivations, and goals – maybe we can install the proper countermeasures to deal with the real threats.
>
> Bruce Schneier, credited with the first documentation of attack trees, *Dr. Bobb's Journal, December 1999*

1.1 Motivation

Organizations face many types of threats to their security. These threats range from physical threats to malicious insiders and cybercriminals [1]. In order to protect themselves against this, organizations often appoint security analysts that are tasked with identifying and implementing the most suitable countermeasures against these threats. But how can a security analyst identify the dangerous threats and select the most cost-effective countermeasures?

This is currently mostly done on the basis of experience or standardized solutions, but such an approach hardly guarantees that the best decisions are made. A security situation is unique for each organization and therefore the prevention measurements should be tailored to each specific organization.

Several recent research projects are focusing their effort on providing security analysts with tools that assist them in obtaining sufficient knowledge about each unique security situation so that they can base their decisions on accurate, organization specific information about the possible threats and (the effectiveness of)

their countermeasures. Examples of such research projects include TREsPASS [2], VISPER [3] and SHIELDS [4].

These projects aim to provide such information by developing tools that can be used to analyze the threats (for example to determine the most critical and/or most probable attacks) and to analyze the countermeasures for such threats (for example by determining the most cost effective countermeasures, or which countermeasures prevent the most dangerous attacks).

At the foundation of many of these research projects lies the attack tree formalism, a prominent security formalism for threat analysis [5]. Attack trees are an approach for representing threat scenarios (i.e. attacks) and their possible countermeasures in a concise and intuitive manner. They allow for a hierarchical decomposition of complex attacks and countermeasures into simple, easily understandable and quantifiable actions [6].

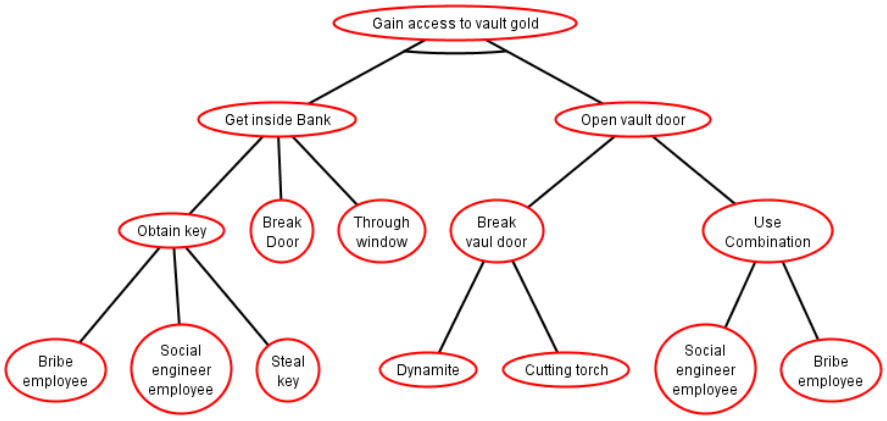


Figure 1.1: An example attack tree representing the different possible attacks to gain access to a bank's gold

Figure 1.1 shows an example of an attack tree that represents the different possible attacks to gain access to a banks gold. The root describes the attacker's goal and its children define how to achieve this goal. The root node has two children and represents an AND, meaning that both its children should be achieved, in this case: getting into the bank and opening the vault door. Each node is then refined until only basic actions remain. The remaining nodes represent ORs, and thus only one basic action is sufficient to achieve each step, for example bribing an employee to get the key for the first step and using dynamite to blow the vault door open for the second.

Attack trees are considered useful for qualitative analysis, i.e. manual analysis, of the different attacks as they often highlight the similarities and differences of attacks in a concise manner, but they are also used for quantitataive analysis by assigning values, such as probability of success or cost, to all the so-called *basic actions* that each attack consists of. This way, the most probable attack can

be calculated; for example, or how certain attack scenarios can be prevented in the cheapest manner [7].

Several recent efforts [5, 6, 8] have focused on developing methods for multi-parameter analysis, such as assigning the probability of success, cost, probability of detection and time required to each basic action, and then determining the most optimal attack scenarios. These efforts add to the ever increasing set of analysis methods for (the attacks represented by) an attack tree.

But while many research efforts are focusing on the analysis of attack trees, the usefulness of these approaches heavily depends on the quality of the attack trees used. For instance whether the attack tree contains all attacks, or if they are on the right level of detail and annotated with the correct information.

Using suitable attack trees that describe and quantify all threat and defense scenarios for a specific organization is therefore of vital importance for successful analysis results.

Obtaining such an attack tree is, however, difficult. Attack trees are traditionally manually constructed by so called Red Teams that consist of security experts, but this manual development is tedious, error-prone and impractical for larger organizations [9]. Attack trees for larger organizations will become large themselves (into the 1000-10.000 basic actions range for example [10]) making it impractical or even infeasible to develop attack trees manually.

Therefore, a different research direction looks into how such attack trees can be generated automatically, as automating the construction process also ensures that the result is *exhaustive* and *succinct*. Exhaustive, as it contains all possible attacks against an organization, and succinct because it will only contain valid, executable attacks. This work aims to provide a contribution to the research generating attack trees.

1.2 Problem Context

Recently there have been a number of research efforts on the topic of generating attack trees (e.g. [11, 12, 13]). Generally all of these efforts are based on the same idea: In order to develop an organization/problem specific attack tree, the approach requires information about the organization/problem in order to determine the possible attacks that it should describe. This information can be specified in the form of a dynamic model that describes the organization/problem.

Generally these previous efforts can be seen a a two-step process: (1) Identifying possible attacks and (2) Constructing the attack tree using this result. The two steps of the process and their input and output are shown in Figure 1.2 are discussed in more detail below:

(A) The input of the attack tree generation process is a dynamic input model. A dynamic input model describes a system or organization through two parts: a static part and a dynamic part. The static part for example

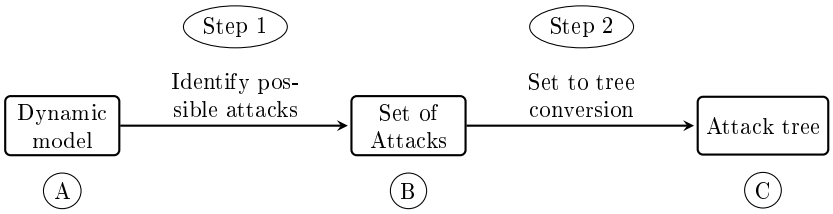


Figure 1.2: Overview of the generic attack tree generation process which consists of two steps, each with its own input and output.

describes the layout of an organization. Figure 1.3 shows an example of the static part of a dynamic model.

The dynamic part specifies the possible interactions of the elements of this static description, i.e. the possible actions an attacker can undertake, when they are applicable and what their effect is. An example of this is that the attacker can relocate to inside the bank by going through the door, but this door requires a key. The attacker should therefore somehow obtain the key to open it, or it could break the door.

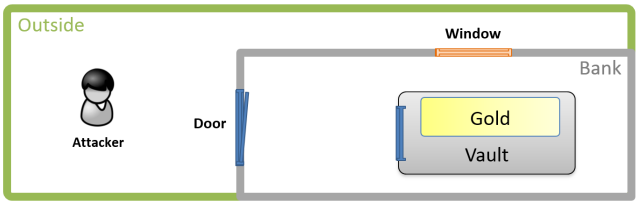


Figure 1.3: An informal visualisation of the static part of a dynamic input model

Step 1 The first process step is to analyse the dynamic input model and identify all possible attacks. This is done by analysing the possible interactions of model elements (e.g. exploring at all sets of attacker actions) in order to identify set of actions that result in a state of the model where the attacker has achieved his goal. Each such set of actions describes an attack.

As an example, Figure 1.4 shows a state of the model where the attacker has reached his target. The task of this step is thus to identify the different sets of actions for the attacker to go from the initial state of the model (figure 1.3) to this state.

B The result of the first step is a set of attacks. For this simple example there are five different actions that can be perform to reach the first step (getting inside the bank) and four options for the second (opening the vault) resulting in a total of 20 (5*4) possible attacks.

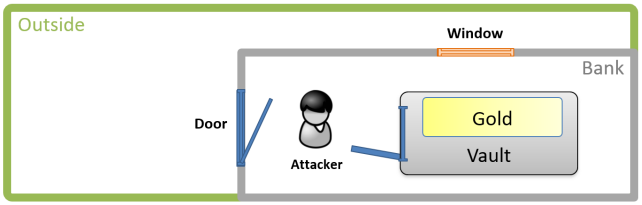


Figure 1.4: An informal visualisation of a model state where the attacker has reached his target: Having access to the gold.

Step 2 The second step in the process is to convert a set of attacks into a single tree representation and optionally making this representation more compact.

C The result of the process is an attack tree that describes all possible attacks that are contained in the input model. The attack tree depicted earlier (figure 1.1) shows all possible attacks that can be identified in the input model (figure 1.3).

The first step of the process, identifying all possible attacks by analysing the input model, has received the most attention as it is considered the most challenging part of the process. On the one hand, this analysis should be exhaustive and succinct in order to guarantee a valid attack tree, but on the other hand the analysis should also be able to receive and analyse large input models in a timely matter in order to be usable in practice.

The second step in general (converting a set of actions into a tree) has also been studied but there exists a simple solution to perform this step. There are however many tree representations of the same set of attacks and finding and optimal version of the tree can a computational challenge.

Therefore, the first step is considered the most vital part of the process, but previous efforts have not resolved the challenges of this step in a satisfactory manner. The following subsections briefly introduce the previous research efforts that have focused on this task and discusses why they are not sufficient.

1.2.1 Identifying attacks by constructing a reachability graph

The basic approach for identifying all threat scenarios for a given organization from its model was first introduced by Sheyner *et al.* [9] who demonstrated that the analysis of a model can be performed by using existing symbolic model checking algorithms that produce a reachability graph.

A reachability graph nodes describe all states of model and its edges represent actions and point to the resulting state of performing an action in a certain state. From this reachability graph, all sequences of actions that result in a

state of the model where the attacker has reached his goal can be determined and therefore all possible attacks can be identified in this manner.

But while they demonstrated that the construction of a reachability graph is sufficient for this task, they noted that the performance of this approach does not scale well for larger input models. This model analysis produces a prohibitively large set of possible attacks, since, as is usual in model checking applications, the reachability graph grows exponentially large for the number of independent model elements [14]. In particular the exploration identifies many duplicate attacks that differ only in the order in which independent attack steps are attempted [10].

Even though there has been significant progress in the underlying concepts of this basic approach, such as in the field of model checking where fundamental data structures such as Binary Decision Diagrams have been investigated and have enabled significant advances in the size of systems that can be analyzed, no model of practical size has yet been analyzed using this basic approach [15, 10, 9].

Because of the scalability limitation of this basic approach, several research efforts have looked into improving the scalability by altering the approach itself, instead of improving its underlying concepts. The two main solution directions that have been investigated are discussed in the following subsections.

1.2.2 Optimizing the analysis for a domain-specific input modeling language

An alternative solution direction has focused on optimizing the analysis method by committing to a domain-specific input modeling language and integrating the concepts of this language into a custom developed model exploration approach.

Such a custom developed exploration approach is often based on dividing the exploration into a set of independent sub-problems and developing directed searches to solve these sub-problems. This allows the approach to avoid having to explore all sequences of basic actions; instead it only has to explore all combinations of solutions to sub-problems. It is therefore able to significantly reduce the exploration required compared to generic, uninformed exploration procedures like the reachability graph approach.

For example: If the input models are known to consist out of locations and ways to move between these locations, the exploration approach can define the movement between locations as a sub-problem and develop a custom directed search to determine the different ways to move between two given locations.

There are however downsides to optimizing the analysis for a domain-specific input modeling language. First of all, it is difficult to develop these custom exploration approaches while still guaranteeing the *exhaustive* and *succinct* properties: For example, it is challenging to guarantee that a custom exploration will not miss any attack possibilities and it is difficult to define independent sub problems as there are often relations between two sub-problems.

The biggest downside to this alternative solution however is that it significantly reduces the reusability of the approach, as changing the modeling language or its concepts will likely invalidate the custom developed exploration approach by altering the possible sub-problems the analysis can be divided into. Therefore, a change to the modeling language will often require a redesign of the analysis method.

This reduced reusability is a problem because the question of how to model an organization on the right level of detail so that it contains all its security properties without overhead is still open. There have been several security frameworks and modeling languages developed recently. One example is the Portunes security Framework [16] which developed a unified model to capture the relations between the physical, digital and social security domains and attempts to extract attacks that span all of these combinations, i.e. find attacks even if their attack steps belong to different domains, such as social engineering a receptionist and breaking a vault door. There are also other recent efforts, such as [17, 18] that have used different concepts.

In addition, is it not expected that there will be a single modeling language that is suitable for all different purposes and security domains but rather that there will be a number of such domain-specific modeling languages for different purposes, which would then require a custom generation approach for each language.

Therefore, in order to avoid having to redesign and redevelop an analysis method for each domain-specific modeling language (alteration), there exists a need for a generic yet scalable analysis method that does not commit to a domain-specific modeling language.

1.2.3 Improving scalablity by assuming monotonicity

Another research direction for improving the scalability of the analysis of a model has focused on using the so called *monotonicity assumption*, which basically states that all attacker actions can only contribute to the attackers capabilities (i.e. that actions can only enable and not disable each other), to reduce the amount of exploration required.

The monotonicity assumption can be used to reduce the exploration required by declaring that the order of performing actions is not relevant as it has no influence on the result. Therefore the exploration only has to find the set of actions required to achieve the goal instead of exploring all possible sequences of actions.

Ammann *et al.* [14] have shown that it is possible to reduce the state-space explosion encountered during regular exploration in a significant manner, while Dimkov [16] has demonstrated that the exploration is more scalable as its computational requirements are bounded polynomially to the number of elements in the model.

Therefore, this assumption allows for a scalable exploration of the model by assuming that actions have no negative effect on each other (e.g. they cannot

disable each other).

It has been argued that this is assumption is natural for certain security domains, such as network security. The authors of [14] argue that in the network security domain, access to digital locations and credentials is in general not lost during an attack, and exploit possibilities are never invalidated by performing other actions during the attack. Although there exist counterexamples, they argue that the benefit of the increased scalability outweighs these exceptions and therefore makes monotonicity a reasonable assumption in the network security domain.

If one were to use a similar modeling approach to model the physical security domain, however, the monotonicity assumption could result in an attacker to have access to certain locations at once (as moving to different location would not remove earlier obtained access), meaning the attacker would be able to be in different physical locations at the same time, which is not a natural assumption to make.

Therefore, the monotonicity assumption for attacker actions does not hold in general for different security-domains, as most actions are not by definition only contributing.

The result of upholding the assumption when it does not apply generally results in an overestimation of the possible attacks. This is because in addition to finding all valid sets of attacker actions that result in a successful attack, the exploration can also find sets of attacker actions that turn out to be in conflict with each other when the assumption is not made [16]. But if the input modeling approach also supports negative application conditions for specifying the possible model interactions, e.g. describing what prevents an action from being executable, upholding the monotonicity assumption may also result in not all attacks being found (as actions will not remove capabilities, which otherwise might make other actions applicable), which is a severe drawback of the approach.

Therefore, the monotonicity assumption is dangerous to uphold when it does not apply, as it can undermines both the *exhaustive* and *succinct* properties that an attack tree generation approach should strive to have.

1.2.4 Problem Statement

The analysis of a dynamic model has proven to be a challenge as it suffers from scalability issues for larger input models. The number of sequences of model element inter-actions/attacker actions that need to be explored increases significantly for each added model element and with it the time needed to find all possible attacks. Standard generic exploration methods that attempt to explore all possible sequences of actions result in a state-space explosion which, for larger input models, quickly makes the model analysis infeasible.

The two main alternative research directions investigated to reduce the scalability issue both contain conceptual drawbacks that make them not suitable as the analysis method of a generic attack tree generation approach.

Therefore, there exists a need for a security-domain independent yet scalable analysis method for dynamic models in order to identify all attacks.

1.3 Research Goals

The objective of this research project is to develop a generic approach for automatically generating an attack tree from a given dynamic input model. In order to develop such an approach a generic yet scalable dynamic model analysis method is required in order to identify all possible attacks the model contains.

Therefore, the main goal of this research is to investigate an analysis method that has the potential to improve the scalability of a approach while remaining (security-)domain independent.

1.3.1 Partial-order reduction

One key insight into the scalability problem is that the state-space explodes because it attempts to explore every possible sequence of actions, even though the order between many of those actions is irrelevant.

This same observation was made by the approaches using the monotonicity assumption. These approaches reduce the exploration to only exploring all possible sets of events (instead of all sequences).

Partial-order reduction, a technique for reducing the size of the state space to be explored by exploiting the concurrency between events, has the potential to take the best of both sides by only exploring the sequential orderings of a set of events when the order is relevant (when events are not concurrent) and otherwise only exploring the set.

The scalability improvement of this technique depends on the events modeled in the dynamic model; loosely speaking, the more concurrent events are contained in the model, the better the scalability improvement is when using partial-order reduction [19].

While the potential of using partial-order reduction to reduce the exploration required in analysing a model has been discussed in related research efforts [10], it has not yet been shown that it can significantly improve the scalability of analysing a dynamic model for all possible attacks.

The main goal of this work is therefore to evaluate if using partial-order reduction can improve the scalability of the model analysis.

1.3.2 Graph transformations

As mentioned previously, committing to a security-specific modeling language and/or committing to domain-specific concepts significantly reduces the reusability of the approach for different/modified input models.

While the approach should not commit to a security-domain specific modeling language, it still requires an input modeling language to define the exploration upon. The proposed solution is to use a high-level generic modeling paradigm for this purpose.

By using a high-level modeling language as input, the approach is capable of receiving models of a large number of different security-domains by expressing these models using the generic modeling concepts of the language. Even existing (security-domain) specific modeling languages and their models can be used as input for the approach in this way, by developing a mapping between the concepts of the specific languages to the concepts of the generic language.

We consider the graph transformation modeling paradigm to be a good choice for such a generic modeling language and therefore want to evaluate it for this purpose.

Graph transformations are considered as a flexible and generic modeling formalism. [20]. One of the advantages of using graph-based modeling is that it combines user friendly, intuitive and visual features with formal semantics and algorithms that allow analysis [21]. In addition, it has already been shown that graph transformations can be used as a modeling language for organizations spanning multiple security domains [16].

There are, however, also additional benefits to using graph transformations as a modeling paradigm.

One benefit is that there exists a large body of research and existing solutions based on the formal semantics of this modeling paradigm, and some of these existing solutions can be reused for the approach developed in this work. One key example of this is an existing approach for exploring graph-based dynamic model using a partial-order reduction technique called unfolding.

Other benefits are the expressiveness of the paradigm, such as supporting negative application conditions for specifying attacker actions. In addition it provides a nice separation of the static and dynamic specification parts of the model, allowing the models to be easily tweaked to alter its problem specification. This makes it possible to alter the capabilities of the attacker with minimal effort for example.

Therefore, the second goal of this research efforts is to investigate if graph transformations are a suitable modeling language that can be used as the input language of the generation approach.

1.3.3 Compact attack tree representation

In addition to identifying all possible attacks from a model, the set of attacks also needs to be extracted and converted into a single attack tree. There are multiple manners in which this conversion can be performed and different ideas on what the optimal tree representation looks like.

Therefore, this work also looks into the different possibilities for converting a set of attacks to a tree representation.

One interesting feature of the partial-order reduction exploration is that this also results in knowledge about when the order of performing actions is relevant and when not or even if two actions can both be performed or not. This additional information can be included in the attack tree in the form of sequential and gates and exclusive or gates. This information can then be used by the analysis tools, for example when calculating timing aspects, to allow for new analysis options of the same set of attacks.

1.3.4 Research Questions

In order to make the described research goals of this work more concrete, the following set of Research Questions were developed:

RQ1: Can graph transformation be used as a modeling paradigm to specify systems and organizations as input models for the attack tree generation approach?

RQ2: Can partial-order reduction, and specifically the unfolding of a graph transformation model, be used to reduce the state-space explosion problem that occurs during the automated exploration of a model?

RQ3: How can the set of attacks be converted into an attack tree, what are the trade-offs and how can additional information such as sequential AND's be included in the tree?

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 will give an introduction of the main concepts on which this work builds, including the graph transformation modeling paradigm, partial-order reduction and attack trees. Chapter 3 discusses the unfolding of a graph transformation system as a specific partial-order reduction technique to reduce the exploration required to identify all possible attacks. Chapter 4 then shows how the set of possible attacks can be extracted from the unfolding and converted into an attack tree.

This is followed up by a two-part evaluation. First Chapter 5 will describe a case study where this work's attack tree generation approach was applied to an existing case study and evaluate its quantitative aspect. This is followed up by the quantitative evaluation in Chapter 6 which will discuss the performance of the unfolding as an exploration method compared to constructing a reachability graph.

Chapter 7 will give a more detailed overview of the related work on the topic of exploring a dynamic model for possible attacks and the construction of an attack tree based on these attacks. Finally, Chapter 8 will discuss the contribution of this works proposed approach, given an overview of the main options for future work and give the final remarks of this thesis in the conclusion.

Chapter 2

Background

> "Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."
>
> Lewis Carroll, *Alice in Wonderland*

2.1 Graph Transformation Systems

Graph Transformation Systems (GTS) are considered as a flexible modeling paradigm that can be used for modeling in a wide array of problem domains because of their underlying data structure, that of graphs, is capable of capturing a broad variety of systems [20].

If a system to be modelled can be described as consisting of entities and relations between those entities, the system is naturally represented by the nodes (for entities) and edges (for relations) of a graph.

When modeling a building layout for example, the nodes could represent different rooms and the edges could represent how those rooms are connected.

When explore a dynamic model of a system or organization, the model must also specify how other configurations or states (of the modeled system) can be reached, for example how relations between entities can be added or altered. This is where graph transformations can be used. Graph transformations specify when and how the entities and relations of a graph model can be altered, removed or created in order to reach different states, i.e. how the graph data-structure can be transformed.

Using graph transformations it can be specified for example how an actor might move between two locations, or how the system reacts to certain situations.

These two notions of a graph as a data structure and graph transformations for specifying the dynamics of a model form the foundation of a GTS, also referred to as a Graph Production System.

2.1.1 The concepts

The GTS modeling paradigm started out from a few simple ideas. Because of application needs and theoretical problems it was later extended with additional concepts and formalisms that have also increased the complexity of the paradigm

In this section, only the basic concepts used in this work are introduced. For a more detailed description of these and the other extensions, we refer to the paper Graph Transformations in a Nutshell [22].

Graph data structure The key concept, the graph data structure, consists of a set of nodes and a set of (directed) edges between those nodes.

Both nodes and edges can also have labels. Edge labels are descriptions that indicate the type of relationship that they model, while node labels can be either an *id*, *type* or *flag*. The *id* label of a node simple refers to its unique identifier, the *type* label is used to indicate the type of entity that a node models, and a *flag* can be used to model a Boolean condition, with a node satisfying a certain condition if it contains the flag. The *flag* is then used as a way to differentiate between two nodes of the same type.

For example, an edge between two nodes could have the label *connectedTo* to indicate that two locations (modeled as nodes) are connected to each other. The node labels then indicate that the two nodes are locations, what their id's are and a flag can be used to give a location a certain property.

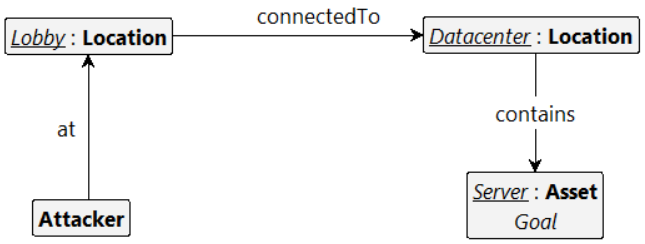


Figure 2.1: Example of a graph data structure modeling a simple data-center

A simple example of a graph, representing a model in the context of the physical security domain, can be seen in figure 2.1. The example graph contains four nodes: Two representing a *Location* entity, one the *Attacker* and the fourth represents an *Asset*. Each node is also labeled with an id, for example the *Lobby* location.

There are three arrows representing relations between the entities. The attacker is currently residing in the *Lobby*, the *Lobby* is connected to the *Datacenter* and the *Datacenter* contains the *Server* asset.

Finally, the *Asset* is annotated with a flag named *goal* to indicate that this asset is the target of the attacker.

Transformation rules The dynamic behavior of a model is specified by a set of graph transformation rules that specify when and how a transformation can be applied.

A transformation rule consists of the following:

- A pattern that must be present in the host graph in order for the rule to be applicable;
- Elements (nodes and edges) to be deleted from the graph;
- Elements (nodes and edges) to be added to the graph;
- Sub-patterns that must be absent in the host graph in order for the rule to be applicable.

Alternatively, one may think of a rule in terms of application condition and modifications, with the elements of the required pattern, the elements to be deleted and the sub-patterns that must be absent describing the application condition, and the modifications being described the elements that are deleted and the elements that are to be added to the graph.

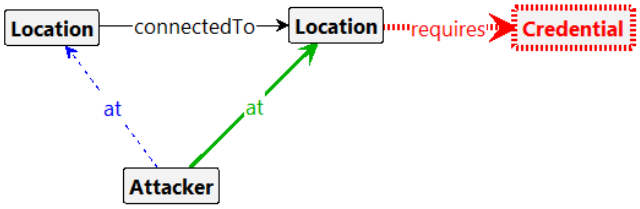


Figure 2.2: Transformation rule example

Figure 2.2 shows a small example of a transformation rule that contains all of these concepts. The transformation rule describes how an attacker can move between two locations.

For clarity, the visualization of transformation rules of [20] is used. Here all elements of a transformation rule are combined into a single graph, using colors and shapes to distinguish them.

The example transformation rule consists of the following elements:

- The black 'reader' elements, in this case the two locations, the *Attacker* and the *connectedTo* edge, are elements that must be present before the rule can be applied and are preserved after the transformation. In our example, this describes that there must be two locations that are connected and there must be an *Attacker*.
- The fat dashed red 'embargo' elements, in this case the requires arrow to the *Credential* entity, describe a pattern that must be absent in the graph before the rule is applicable. In the example, this describes that

the second location should not require a *Credential* (such as a key) before it can be accessed.

- The thin dashed blue 'consuming' elements, in this case the *at* edge from the *Attacker* to the *Lobby* location, describe the elements that are deleted after applying a rule (and are required to present before applying it). In the example, this describes that the *Attacker* should be at the first location, but this relation is removed after applying the rule.

- The green 'creator' elements, in this case the *at* arrow to the second location, describes the elements that will be added in the transformation. In this example, it describes that after applying the transformation, the *Attacker* will reside at the second location.

The overall effect of the rule is to search for two connected locations, where the *Attacker* resides at the first location and the second location does not require a *Credential*. If it finds such a place in the graph, the graph is then transformed by moving the *Attacker* to the second location, which is done through deleting the first *at* relation and creating a second *at* relation.

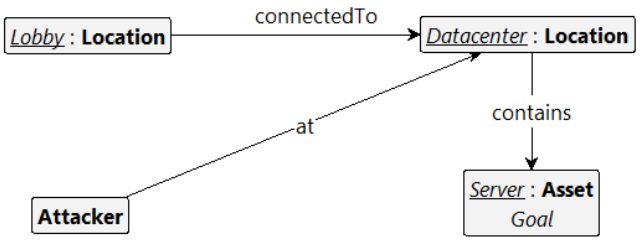


Figure 2.3: Transformation rule application example

Figure 2.3 shows the result of applying the transformation rule to the example graph (figure 2.1) introduced earlier. The effect is that the *Attacker* now resides at the *datacenter* location.

Finally, it is important to understand that depending on the input graph, a transformation rule may be applicable to different places in the graph. In addition, a GTS could contain a set of different transformation rules. Therefore, using an initial input graph, a set of transformation rules can be used to find different configurations or states of the graph.

Type graph One of the many extensions of the modeling paradigm is the use of a type graph (TG). Without a type graph, graphs can be arbitrary designed, e.g. there are no constraints on the allowed combinations of nodes, edges and their labels. When using a type graph, a special graph is designed that describes all valid graph instances, i.e. all combinations of nodes, edges and labels that are possible.

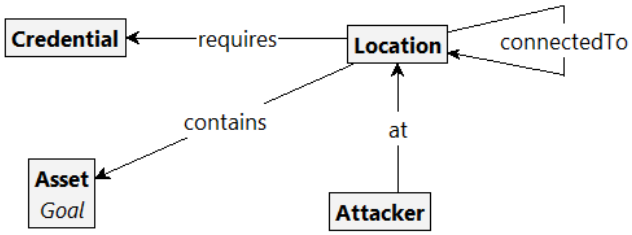


Figure 2.4: Example of a type graph

Using a typed GTS, a graph must be well-typed, indicating that the graph elements must be mappable to the type graph. It can be statically verified if the initial graph is well-typed and each graph transformation maintains this well-typedness.

A simple example of a type graph, for which the earlier examples are well-typed, can be seen in figure 2.4. The type graph describes valid combinations of nodes and edges, or entities and their relations. The example type graph first describes all valid node types, in this case: *Location, Attacker, Credential, Asset*.

The type graph also describes the valid relations between those types: *Locations* can be *connectedTo* other *Location*s, an attacker can be located *at* a *Location* and the *Location* may require a *Credential* and contain an Asset. Finally, an asset node may contain a *goal* flag, indicating that obtaining it is the goal of the *Attacker*.

It is important to also understand the limitations of a type graph. While it can describe what valid nodes and edges are, it cannot specify if certain graph elements should be present, or in what number. It is not possible to specify that there may only be one *at* arrow in the graph. This property must be preserved through the transformation rules.

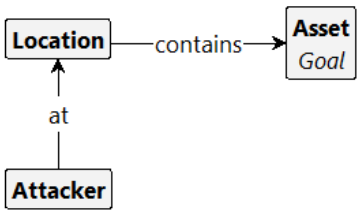


Figure 2.5: Example of a condition rule that specifies when the attacker has reached his target

Condition rule (Goal state) In addition to regular transformation rules, it is also possible to specify a condition rule. A condition rule is a rule that only specifies an application condition and not a modification and can be used

to determine if the graph satisfies a certain condition. It can specify a state of the graph that is interesting, for example a state where an attacker has reached his goal. If a graph satisfies the application condition of such a rule, the graph describes the interesting state.

These type of rules can be used during the (automated) exploration of a GTS to determine when a successful chain of rule applications has been found that results in a state that satisfies the specified condition (e.g. a state where the attacker has reached his goal).

Figure 2.5 illustrates a possible condition rule for the example model. It specifies that the *Attacker* has reached its target, if it is at a *Location* that contains the *Asset* the *Attacker* is after.

2.2 Automated exploration

Given a system or organization modeled in a GTS, it is then possible to explore the behavior of this model by applying transformation rules and registering what states are reached and what other rule applications become available.

In this manner, the GTS can be explored to find (routes to) interesting states, such as goal states.

However, the interleaving of all rule applications results in a large amount of non-determinism which makes a manual search for possible attack sequences infeasible. Therefore automated search is necessary to systematically find all rule application sequences and goal states [23].

This section will first describe the commonly used method to describe the result of an exploration, namely a state-space. Following this it will be discussed how an automated exploration of a GTS can be performed.

2.2.1 State space

The default result of an exploration is a state space. A state space describes all possible states and which states are (directly) reachable from other states. From the state space it can be derived what all the possible sequences of rule applications are.

Let us demonstrate it with an example. The start graph is altered to a model where the second *Location* requires a *Credential*. Then there are two ways to move to the second *Location*, one is by faking the *Credential*, and the second is by first obtaining the *Credential* and then using it to move.

Exploring these three production rules and the simple start graph result in a relatively small state-space, see figure 2.6. The exploration starts from the start graph, or state s0. Two rules can be applied on this graph, the FakeCredential rule and the ObtainCredential. Applying these rules results in different states, s1 and s2 respectively. From the graph state s1, it is then still possible to apply the ObtainCredential rule, resulting in state s3. Similarly, from s2 both FakeCredential and UseCredential can be applied, both transformations also

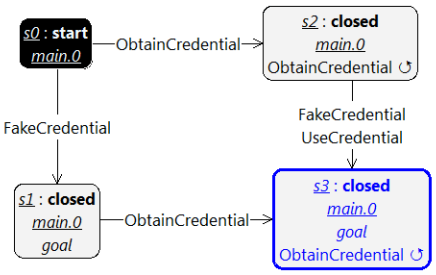


Figure 2.6: State space resulting from automated exploration for a small input graph

resulting in s3. The idea behind this is that both transformations move the attacker to the second location. Whether the actually obtained credential is used or not, does not matter according to the model.

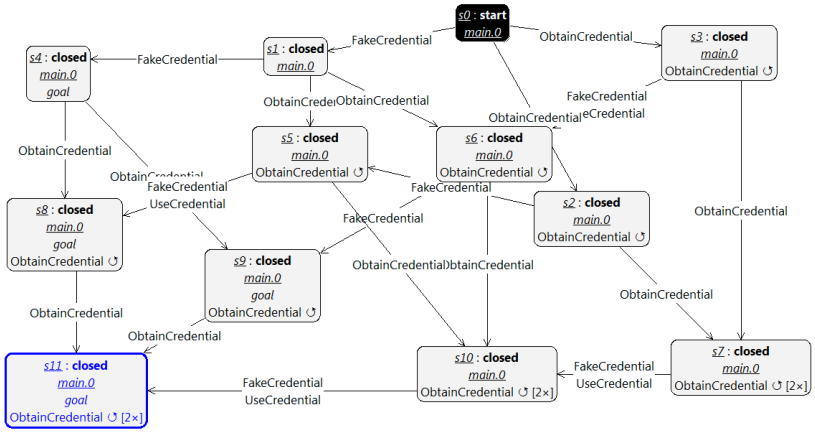


Figure 2.7: State space resulting from automated exploration for a slightly larger input graph

While the state space for this simple example is still small and intuitive to understand, this quickly becomes a different story when larger input models are explored. If we were to add a third location that requires an additional credential to the start graph of the this example, state space already increases significantly, see figure 2.7. The interesting thing is not the state space itself but its size, compared to the previous example.

The issue is that state space does not scale well compared to the size of the input model, as it describes all possible sequences of applying transformations to the start graph. This is also referred to as the state space-explosion problem [23]. Therefore, the size of the state space may increase significantly for each added node to the graph or each added transformation to the GTS.

The result of this issue is that the exploration of a GTS and the construction of its state space quickly becomes a infeasible for larger input models, as is discussed in the following subsection.

2.2.2 Exploration strategies

The automated exploration of a GTS requires a strategy in order to find all sequences of rule applications. Such a strategy could simply be a breath-first or depth-first strategy.

In the case of a breath-first strategy, first all possible transformation rule applications for the start graph would be explored, developing the start of all application sequences. In the case of the depth-first exploration strategy, the exploration would first find a maximum application sequence, before looking into different sequences.

While one of the two strategies might be the best choice considering the different memory and computational requirements of the two, the end result of both strategies is the same complete state-space, and both strategies suffer the same scalability issue as the state-space itself. Therefore, the construction of the state space using these traditional strategies quickly becomes infeasible for larger input models.

There are other possible exploration strategies that attempt to avoid such an explosion, such as linear, random linear and conditional exploration that allow simulation without covering all states, but these do not guarantee that all possible (goal) states are found, as the simulation often only finds a subset of the possible (goal) states of the model.

In this case, if the model describes an attacker and a building, the exploration might only find a subset of the possible attacks. This is a severe drawback of these exploration strategies, as the goal of the proposed artifact is to construct an attack tree describing all possible attacks. Therefore, these exploration strategies are not suitable for identifying all possible attacks. There are other purposes where only finding a subset of solutions might be sufficient, for example to find a solution to a game, or to determine if deadlocks are possible.

Therefore, instead of using a default exploration (or simulation) strategy, an additional exploration strategy is required to reduce the state-space (explosion problem). We propose to use an alternative state-space representation in the form of a partial-order reduction state-space.

2.3 Partial-order reduction

2.3.1 Intuition

As mentioned in the previous chapter, a key reason for the state-space explosion during the exploration of a dynamic model is that it attempts to find all sequences of events, or arbitrary interleavings, in which transformation rules can be applied to a given model.

Since transformation rule application events do not necessarily depend on each other and might even be independent of each other and therefore model concurrent actions, exploring all arbitrary interleavings is not always beneficial.

This problem of investigating the arbitrary interleavings of current actions resulting into a state-space explosion is well recognized and studied in literature and it has been observed that substantial increase in efficiency could be obtained if the enumeration of all possible interleavings is avoided. Therefore several techniques based on partial ordering have been developed to reduce the state-space explosion [23].

It is however important to note that the amount of reduction of the state-space explosion that is obtained when using partial order is dependent on the specific model: If the model contains a large number of independent transformation rules, then partial order reduction can have a big effect on the efficiency. It can also be the case that all transformation rules depend on each other, in that case the worst-case complexity is the same as the default exploration.

Let us demonstrate this concept with a simple example. We define four actions: a1, a2, a3 and a4. Actions a1, a2 and a3 can all be executed directly from the start and are concurrent, i.e. executing one will not effect the others. Action a4 however depends on a1, a2 and a3 being executed before it can be executed itself, e.g. it consumes elements created by the other actions.

A regular exploration would explore all valid sequences of events, events, in this case:

- a1,a2,a3,a4
- a1,a3,a2,a4
- a2,a1,a3,a4
- a2,a3,a1,a4
- a3,a1,a2,a4
- a3,a2,a1,a4

A partial-order reduction exploration would only produce the relations between these actions. For example:

- a1
- a2
- a3
- (a1 + a2 + a3), a4

Stating that a1, a2 and 3 do not have dependencies, but a4 requires all of them to be executed.

The larger the set of concurrent actions becomes, the bigger the difference between the size of the full exploration and the partial-order reduction overview.

2.3.2 Unfolding

The unfolding a GTS is a technique that can be used to obtain a partial ordering of the transformation rule applications of a given graph transformation system.

The unfolding of a GTS represents a single branching structure that describes all possible transformation rule applications and their dependencies. This structure can be used as an alternative (more compact) state-space representation.

As mentioned: The main difference of this alternative state-space representation compared to a regular state-space is that the regular state-space models all possible sequences of transformation rule applications, while the unfolding structure only models the dependencies of transformation rule applications on other transformation rule applications, thereby (potentially) reducing the number rule application orderings that needs to be explored.

The idea of unfolding a model to achieve partial-order reduction was first developed by McMillan [23] and developed for "plain" Petri nets.

Petri nets [24] are a formal tool/model used for the specification of the behaviour of concurrent systems. At it's basis, Petri nets describe transitions that consume and create tokens to model actions.

It was observed by Ribeiro [25] and Baldan *et al.* [ref] that Petri nets can be regarded as graph transformation systems that act on a restricted kind of graphs that only consume and create elements, or similarly, graph transformation systems are a proper generalisation of classical models of concurrency such as Petri nets. Their works have proposed a transfer of the unfolding construction technique from Petri nets to finite state graph tranformation systems and they have defined it's the functorial semantics [26]

This inital specification of the unfolding concepts for graph transformation systems was only defined for a restricted kind of graphs. The main limitations being finite state graph transformation systems without reading edges or negative application conditions.

The unfolding technique has however attracted considerable attention in the Petri net domain and has been further analyzed and improved, see [27] for an extensive survey and [28] for the corresponding book.

The technique has been extended to Petri nets with read arcs [29, 30] and Petri nets with inhibitor arcs [31]. The technique for nets with read arcs has and inhibitor arcs have both been transfered to graph transformations [32, 33]

These foundational papers are light on implementation details of the technique, but more recent efforts have focused on developing an efficient implementation of the unfolding technique for Petri nets with read arcs [34, 19]. This work has transferred the concepts of this implementation to an implementation for graph transformation systems.

A framework for the unfolding of infinite-state graph transformation systems has also recently been proposed by Baldan *et al.* [35] which is based developing an approximate unfolding with arbitrary accuracy.

In addition to using the unfolding of a graph transformation system to study the concurrent behavior of a modeled system, the unfolding approach has also

been used for generating test cases for code generators [33]. Here the working of the code generators are formalised using graph transformation systems and suitable test cases are systematically derived from it's unfolding.

Using the unfolding This works main research question is to determine if the unfolding of a graph transformation systems can be used as a more scalable exploration method to identify all possible attacks in comparison to constructing a reachability graph.

In order to determine this, this research effort focuses on developing an efficient construction approach of the unfolding of a GTS. Formally, the unfolding of a GTS is based on converting the GTS into a nondeterministic occurence grammar by recording all possible rule applications as events of this grammar and the effects of each transformation rule as items of the type-graph of this grammar.

This works unfolding approach is based on the most recent overview of the work performed on unfolding graph transformation systems and it's semantics [36]. The unfolding approached in described in a much more detailed (declarative) manner in chapter 3. For a more formal definition of the basis of this approach we refer to Baldan *et al.* [36].

Once the unfolding of a model has been constructed, a partial-order on the set of attacker actions for each possible attack can be retrieved from this unfolding and be used to construct the attack tree.

2.4 GROOVE

This research effort uses and extends GROOVE[1]. Groove is a tool for modeling and analyzing Graph Transformation systems developed by Arend Rensink *et al.* [37] at the University of Twente.

Its main features are a Graphical User Interface for modeling a GTS and a set of features to explore (the behavior of) the GTS, see figure 2.8 for a screenshot. For the modeling part, it provides functionality to specify a type-graph, start-graphs, transformation rules and condition rules. It provides static verification of well-typedness for all start graphs and rules.

It also provides broad support for the exploration and analysis of a GTS. This can be performed both manually, by selecting what possible rule applications should be applied and seeing its applied result, or automatically, by selecting the exploration strategy

Through the years the basic functionality has been extended several times, for example with concepts to increase the expressiveness of the graph transformation modeling paradigm or those that reduce the specification size, such as element attributes and control blocks (which specify the order of rule applications). In addition, it has also been extended with numerous alternative methods for the exploration and analysis of a modeled GTS. It does however

[1] http://groove.cs.utwente.nl/

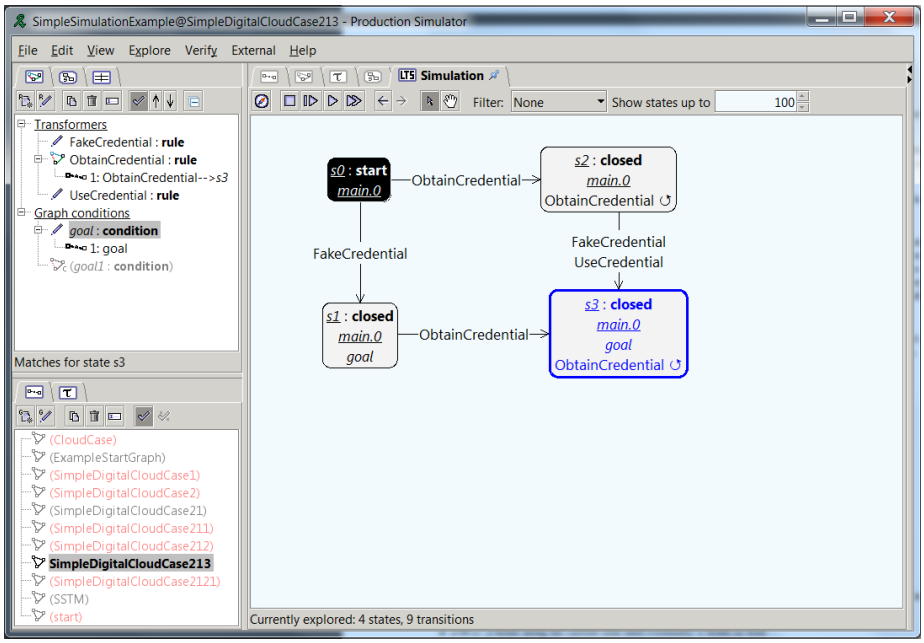


Figure 2.8: Example of GROOVE's GUI for modeling and analyzing GTS

not contain an implementation for an alternative state-space representation that supports partial-order reduction.

In this project GROOVE is used for developing GTS by using its GUI and to explore the developed GTS through it's default exploration. The works approach is developed as an extension of GROOVE that builds on the basic GTS implementation by adding a new exploration strategy (on the basis of a partial-order reduction) and functionality to retrieve and convert the results of this strategy into a attack tree.

2.5 Attack Trees

Bruce Schneier introduced the concept of attack trees based on the idea that to answer questions about the security of a system or organization, one needs to understand the threats [38]. By modeling the threats against a system or organization, one can study and try to understand all the possible attacks and design countermeasures to thwart those attacks.

The attack tree formalism (formalized in [39]) is a graphical security modeling and analysis approach that aims to model the different ways an attacker might reach his goal. The formalism focuses on building a tree where the top node represents the *attacker's goal*. This root node has children that describe all possible attacks that might result in this goal. Each of these children is then recursively refined to describe the different sub-attack of an attack. These

sub-attacks are refined until only atomic actions remain as the nodes without children (i.e. leaf nodes).

Attack trees also indicate the relation between every node and its child nodes, or an attack and its sub-attacks. A conjunctive relation indicates that all child nodes should be achieved in order for the parent to be achieved, while an disjunctive relation means that only a single child needs to be achieved in order for the parent to be achieved.

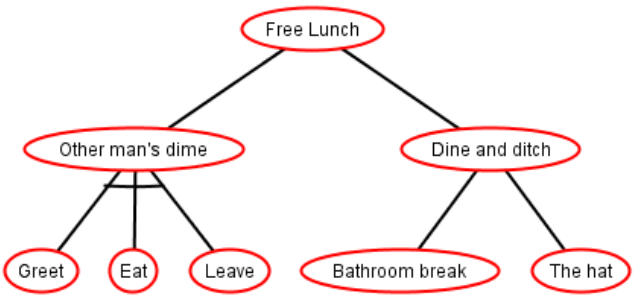


Figure 2.9: Example of an attack tree with different attacks and relations.

An example of an attack tree with different relations as visualized by ADTool [40] can be seen in figure 2.9.

The example attack tree has its as root node, or as the attacker's goal, to obtain free lunch. It specifies that there are two attacks that can result in this goal, and that achieving only one of these suffices. The first attack, letting somebody else pay, consists out of three parts or atomic actions: great, eat, leave. Each of these three needs to be achieved before the attack is achieved (the basic attack tree formalism can not require that children are achieved in order, but there are extensions that add such an order). The second attack, leaving without paying, can be achieved in two manners, disguising oneself through a hat or leaving directly after a long bathroom break.

Given an attack tree, quantitative analysis can be performed for a better understanding and prioritization of the different attacks. One method for such a quantitative analysis attaches a certain value, such as a percentage, that corresponds to a certain domain, such as the chance of success, to each leaf node. The domain then describes how to propagate the values of the child nodes to a parent node, based on the different relations between parent and child nodes (conjunctive, disjunctive and countermeasure).

Figure 2.10 shows how such quantitative analysis might work. Each atomic action (yellow node) is assigned a probability of success. These values, in combination with the relation towards the parent (an AND or OR gate) are then used to compute the probability of success of the of the parent. This probability is propagated upwards until the root probability is calculated.

In addition to the basic attack tree formalism introduced so far, there is a large amount of research on extensions and improvements of this formalism.

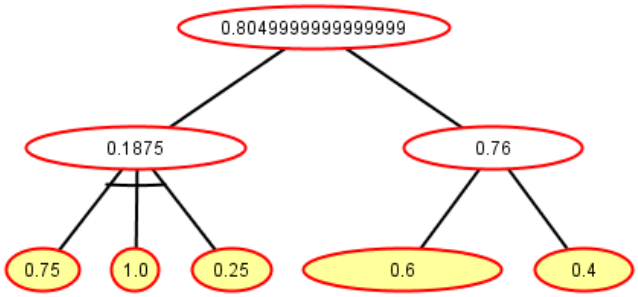


Figure 2.10: Example of quantitative analysis performed on an attack tree

An overview of this can be found in [21]. Two particular interesting examples of improvements are attack trees that share common subtrees, i.e. different branches with the same subtree reuse the same definition of this subtree, and trees with additional gates such as a sequantial AND gate (specifying that all it's children are achieved in order) and an exclusive OR gate (specifying that only one of the children can be achieved). These two improvements are used in this thesis to reduce the size of the generated attack tree and to add additional information to the generated attack tree.

Chapter 3

Part 1: Constructing the unfolding

War is the unfolding of miscalculations.

Barbara W. Tuchman

As a quick recap: The main procedure of an attack tree generation approach is to explore a given input model in order to identify and extract all possible attacks that should be contained in it's constructed attack tree. A default/general-purpose exploration of the model attempts to develop a reachability graph describing all possible states of a model. Constructing a reachability graph, representing the result of the exploration, is known to suffer from a state-space explosion, making it computational expensive to construct and extract information from it for larger input models.

The main goal of this research is to determine if the unfolding of a model can be used as an alternative exploration that reduces the state-space explosion by producing an alternative state-space representation based on exploiting the inherently concurrent nature of actions represented in the model in order to reduce it's size, i.e. using partial-order reduction.

In order for the alternative state-space representation to be suitable for the purpose of exploring a model for all possible attacks, it should (i) be efficient to construct, (ii) contain all required information to identify and extract all possible attacks and (iii) allow for this information to be extracted efficiently.

To determine if the unfolding of a model fulfills these requirements, an unfolding construction is implemented in order to evaluate it for this purpose.

This chapter describes the unfolding construction of a GTS, as is shown in figure 3.1, while the following chapter describes how all the possible attacks are extracted from the constructed unfolding.

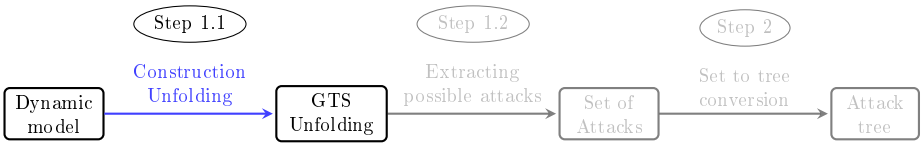


Figure 3.1: This chapter focuses on the step 1.1, constructing the unfolding of a GTS

3.1 The unfolding procedure

3.1.1 Preliminaries

As was explained in the background, the two main component of a GTS are its start graph and the set of transformation rules.

To understand the unfolding procedure, it is important to understand that the elements of a basic transformation rule (both nodes and elements) can be divided into three different sets.

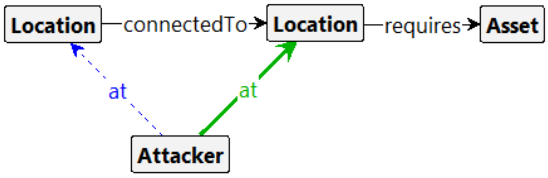


Figure 3.2: *Break Door* transformation rule

Figure 3.2 shows an example of a basic transformation rule which contains elements of all the three sets:

Black The set of *reader* elements. Elements of the transformation rule that must exist before the rule can be applied, but are not altered when applying a rule.

Blue The set of *consuming* elements. Elements of the Graph that must exist before the rule can be applied and are deleted during the application.

Green The set of *creator* elements. Elements that are created after the rule is applied.

3.1.2 Unfolding exploration

Regular state-space exploration During the regular exploration of a GTS, the transformation rules are applied to the startgraph to study their effects. Applying a rule may however make other rules inapplicable if they are not current rules, for example because the applied rule removes certain graph elements the other rule requires. Therefore, when exploring all possible rule applications

of a certain state of the Graph, it is not possible to simply apply all of them one after another, as the non-concurrent rules have an effect on each other. Therefore, after exploring and recording the effects of applying one of the possible rule applications, the original state of the graph must be restored in order to explore the other possible transformation rules, and restoring the state of the graph can become a costly effort, both computational and memory wise.

The Gleuing operation The basic step of unfolding a GTS is to glue all possible transformation rules to the graph in order to study their effect, instead of applying the rules. The difference between gluing a transformation rule to a graph instead to applying it boils down to two points:

1. When gluing a transformation rule to a graph, the set of consuming elements is not removed from the Graph, but the creator elements are added to the graph.

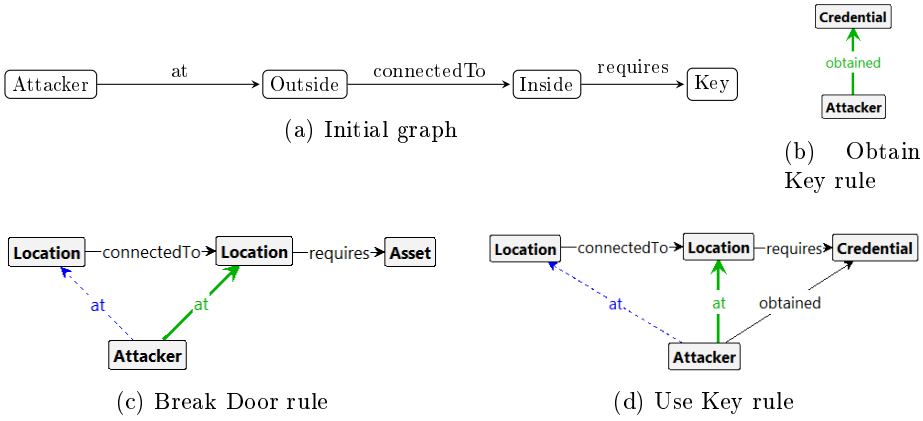
2. All elements of the created set are annotated with the match of the transformation rule that created them. The effect of this is that when two transformation rules create a similar element, such as the same edge between two nodes, both elements are created and unique based on the annotation of the rule that created them.

It is important to observe that because of these differences, the gluing operation of all possible transformation rules becomes concurrent, i.e. the order in which the transformation rules are glued to the graph does not have an effect on the outcome. Therefore, in contrast to regular application where the state of the graph needs to be restored after each single application, the gluing of all transformation rules can be done without restoring the state of the graph.

The gluing operation is used in the Unfolding construction in order to explore the effects of a certain transformation rule application, i.e. to determine what other rule applications are made possible by the transformation rule application.

Exploration procedure The unfolding is constructed in iterations using a breath-first principle. Using the start graph as the input of the first iteration, each iteration determines all possible rule application for its input graph and then glues all these rule applications to the graph in a random order. After performing all glue operations, the resulting graph is then given as the input of the following iteration (which will try to use the effects of the previous glue operations to find new rule application possibilities).

During each iteration, the procedure will also find rule application possibilities that have already been applied in previous iterations, as the reader and consuming set of these rules have not been removed. However, gluing these rules a second time does not have an effect on the graph, as the produced elements are the same elements as the ones produces by the previous gluing of the same rule application, as they have the same rule application as annotation.



(a) Initial graph

(b) Obtain Key rule

(c) Break Door rule

(d) Use Key rule

Figure 3.3: The example GTS setup

The exploration procedure is halted after an iteration does not find new rule application possibilities. In this case, all rule application possibilities have been found.

Unfolding state-space In order to get an actual state-space representation and obtain all dependencies between rule applications, all gluing operations are recorded, including references to all elements of the graph involved with the rule application, i.e. the set of reader elements, the set of consuming elements and the set of creator elements.

At its basis, the unfolding construction comes down to recording all possible transformation rule applications for a certain startgraph. Then, using all of these recordings, the dependencies between rule applications can be determined by comparing the sets of elements of different rules. For example by detecting that a certain rule application has an element in its reading set that another rule application has in its creator set, it is found that the first rule application depends on the second.

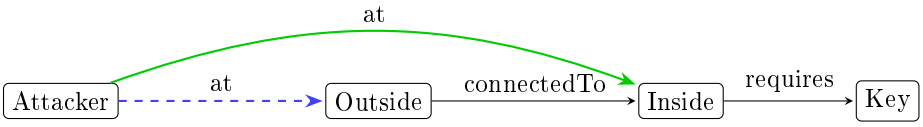
3.1.3 Running Example

Let us clarify this basic procedure of the unfolding construction by constructing the unfolding of a simple example.

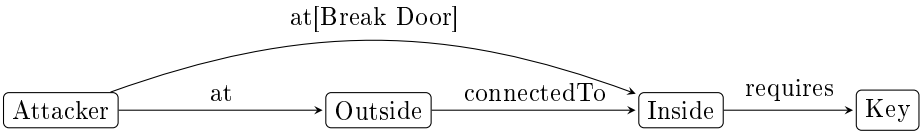
Setup The example GTS used as the running example is shown in figure 3.3. The start graph of this GTS is shown in figure 3.3a and describes an *Attacker* who resides at the *Outside* location. This location is connected to the *Inside* location, a location that requires a key to move to it.

In addition to this graph there are three simple transformation rules, these are *Break door* (figure 3.3c) which simply moves the attacker from outside to inside without using the key, *Obtain key* (figure 3.3b) which provides the attacker

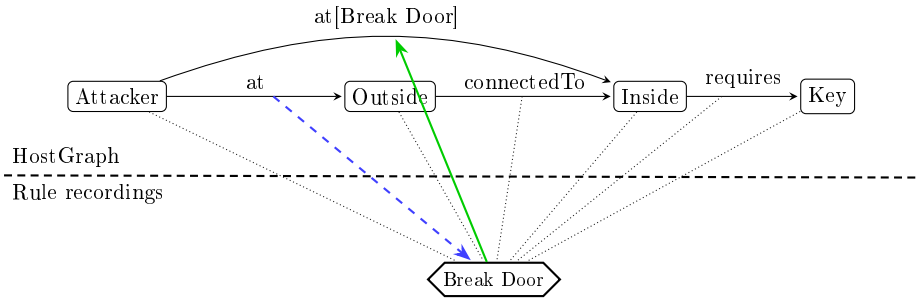
with the key and *Use Key* (figure 3.3d) which makes the attacker use the key to move to the inside.



(a) Visualisastion of a rule application match for the *Break Door* transformation rule.



(b) The result of gluing the rule match to the graph. Only elements of the creator set are added and annotated with the transformation rule.



(c) The result of recording the gluing of the rule match where the recorded rule application references to graph elements that are in its reader, consuming and creator set (references depicted by dotted black, dashed blue and green lines respectively)

Figure 3.4: The process of a basic unfolding step

First iteration The first iteration receives the start graph as its input graph. The iteration starts with determining all possible rule applications for this input graph. For this input graph there are two rule application matches: *Break Door* and *Obtain Key*. The next step is to glue all rule applications in a random order to the graph, as is demonstrated for the *Break Door* transformation rule in figure 3.4.

Figure 3.4a visualizes the match found for the *Break Door* rule. All black elements are in its reader set, the blue arrow in its consuming set and the green arrow in its creator set.

Figure 3.4b then shows the result of gluing this transformation rule match to the graph. The element in the creator set of the match are added to the graph and annotated with the match of the transformation rule application (in this case a match of *Break Door*).

All gluing operations are recorded (externally from the graph) to form the basis of the alternative state-space representation generated by the unfolding. The recording of the transformation rule gluing is shown in figure 3.4c. It shows that the recording of rules is stored separately from the graph, but it references to graph elements that are in reader, consuming and creator sets.

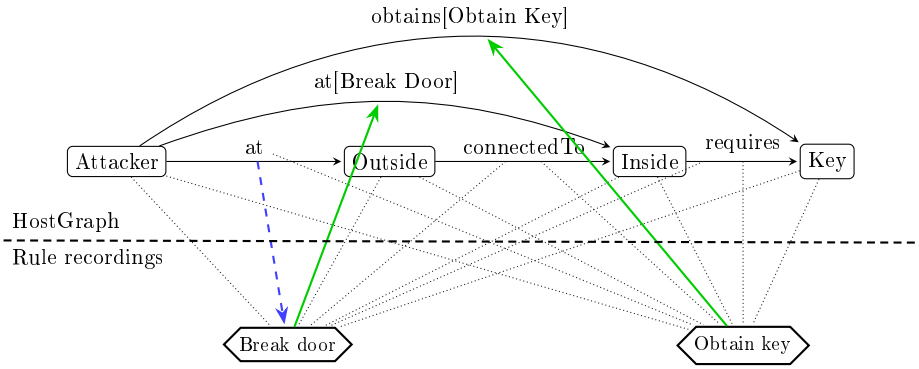


Figure 3.5: Unfolding after the first iteration

Using the resulting graph, the other transformation rule match (*Obtain key*) can be glued (and recorded) in a similar manner to obtain the result of the first iteration of the unfolding, which is depicted in figure 3.5.

The result of gluing this transformation rule is slightly different as the transformation rule does not have any elements in its consuming set, the recording therefore only references to its reader and creator elements.

Second iteration After the first iteration as completed, the unfolding construction will use its result as the input for the second iteration. One again the first step is to find all the matches of all transformation rules. If there are no new matches found then the exploration is terminated, otherwise it continues. In this case one new transformation rule match is found, namely a match for the *Use Key* transformation rule.

The unfolding construction then proceeds to glue this new match the graph and records this process, the result of which can be seen in figure 3.6. One thing to observe is that the *Use Key* transformation rule creates a similar element as the *Break Door* rule, namely an arrow between the attacker and the location Inside. Because each of the created arrows are annotated with their transformation rule match, both arrows are unique and added to the graph.

As mentioned previously, no elements are actually deleted from the graph. Therefore the process of finding matches for all transformation rules will also

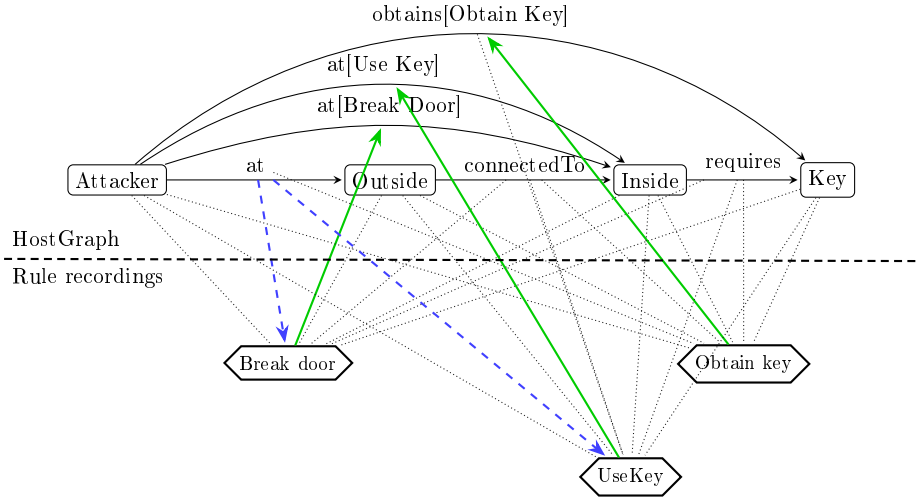


Figure 3.6: Unfolding after the second iteration

find matches that have already been glued to the graph, such as the matches for *Break Door* and *Use Key*. However, gluing these elements a second time (and recording this process) has no result on the graph, as this operation produces the same elements and recordings as the first time. I.e. the created elements are in both cases annotated with the same transformation rule match and are therefore not unique.

3.1.4 The interpretation of unfoldings

By recording the transformation rule gluing during the unfolding it is then possible to infer relations between rule applications. For example, there is a dependency between the *Obtain key* and *Use Key* transformation rule matches. The *Obtain key* rule creates an element that is in the reading set of the *Use Key* set, i.e. it must exist before the *Use Key* rule can be performed. In other words: the *Obtain key* rule must be applied before the *Use Key* rule can be applied.

Larger Example The previous example demonstrated the basic concepts of the unfolding construction. The example was purposefully kept small so as be easy to understand and visualize. On this small scale some of the effects of the unfolding are lost, such as when there are multiple matches of the same transformation rule in the graph. Therefore a slightly larger example GTS is unfolded in order to comment on these additional effects that are noticeable on a larger scale.

The setup of the larger example GTS has the graph depicted in figure 3.7 as its startgraph but the same transformation rules as the previous example. The difference with the previous example is that this startgraph has three locations,

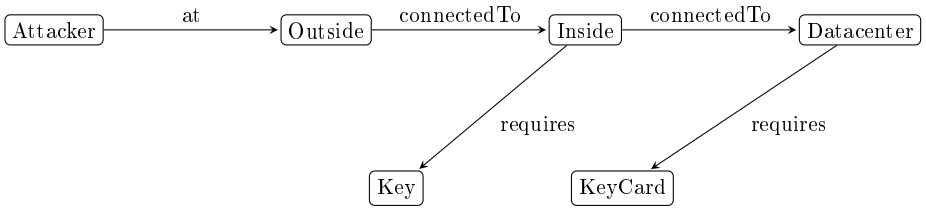


Figure 3.7: A slightly larger startgraph

two of which require a credential.

The complete unfolding of this larger example is shown in figure 3.8. While this figure may be a bit daunting to grasp, it is useful to understand it as it shows the effects that previous gluing operations/iterations have on following iterations. (To reduce the complexity of the figure, all context lines between transformation rules and elements of the initial graph have been removed, just as all element annotations have been removed). To highlight the relationships between the recorded production rules, figure 3.9 shows an alternative visualisation of the same unfolding, focusing only on the elements in the reader, consumer and creator set of each recorded rule application during the unfolding.

Interpretation The main difference with the earlier example is that there are now three locations. Therefore the unfolding does not only explore the ways to get from location *Outside* to location *Inside*, but also how to get to location *Datacenter*. As there are two transformation rules that have the effect of moving the attacker from a certain location to another (connected) location, this gives a total of four different sequences of applying all transformation rules to get from location one to location three. All four of these sequences are shown in the unfolding.

The interesting thing to observe is that, because of the annotation of created elements, there are two 'at' arrows created between the attacker and the second location (Inside). Therefore, each of the two transformation rules that can be used to move between two locations then has two matches (one for each at arrow) in the graph to move to the third location. The intuition behind this is that it can depend on either of the ways to get to the second location. Because each of these matches creates a unique 'at' arrow is created, as each created element is annotated with the specific match of the transformation rule, this results in a total of four 'at' arrows between the Attacker and the third location, with each of these arrows having a unique dependency order on how they are created.

Each of these four 'at' arrows to the *Datacenter* has a unique causal history. The example history of the *at-4 edge* for example is shown in figure 3.10a. This causal history of an element describes all the elements and production rules applications that were involved in creating this element.

From this causal history it can be determined what production rule application this element depends on to be created and in what (partial-) order these

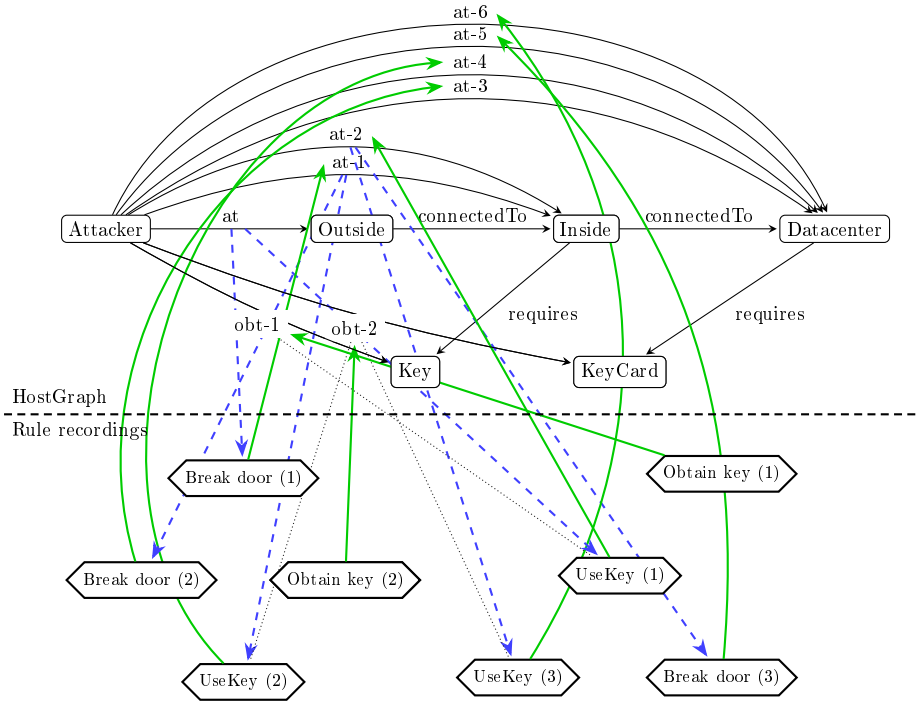


Figure 3.8: Complete unfolding of the larger startgraph. To reduce the complexity, all context lines to elements of the initial graph have been ommited and the annotations replaced by numbers.

rule applications need to be applied. In the case of the *at-4* element, the set of production rule applications is {Break Door-1, Obtain key-2, Use key-3}, and the partial-order dictates that Use key-3 can only be applied after the other two have been applied, but the order in which the other two are applied is not relevant.

We refer to this information about each element as the elements configuration. The configuration of the *at-4* element is shown in figure 3.10b.

3.2 Filtering conflicts

A key ingredient of the unfolding construction is the gluing operation. The fact that the gluing operation does not remove any elements from the graph makes the exploration of the GTS very efficient as it does not have to restore the state of the graph whenever there are multiple rule application possibilities for a certain graph state, which is in contrast to when rules are actually applied during the exploration.

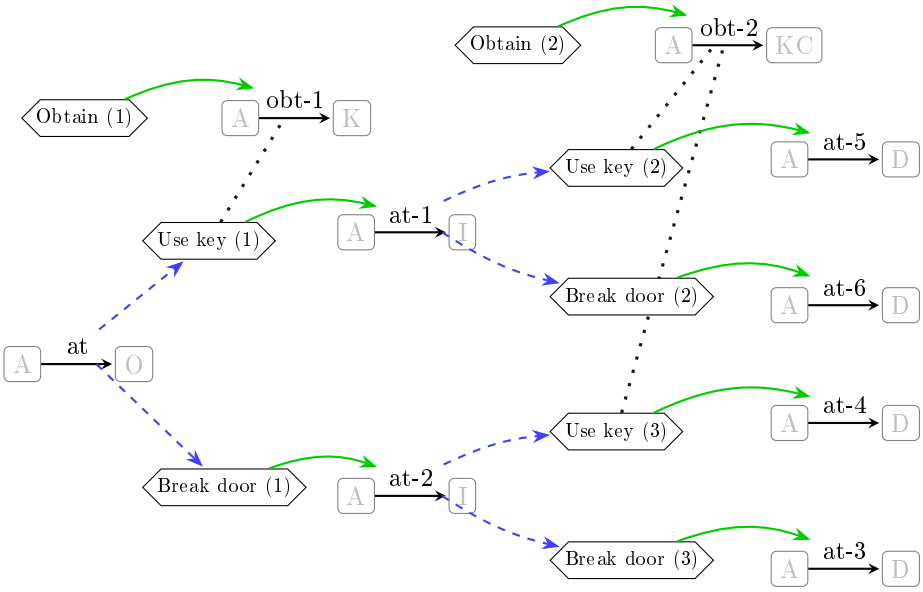


Figure 3.9: Alternative visualisation of the unfolding to highlight relations between rule applications

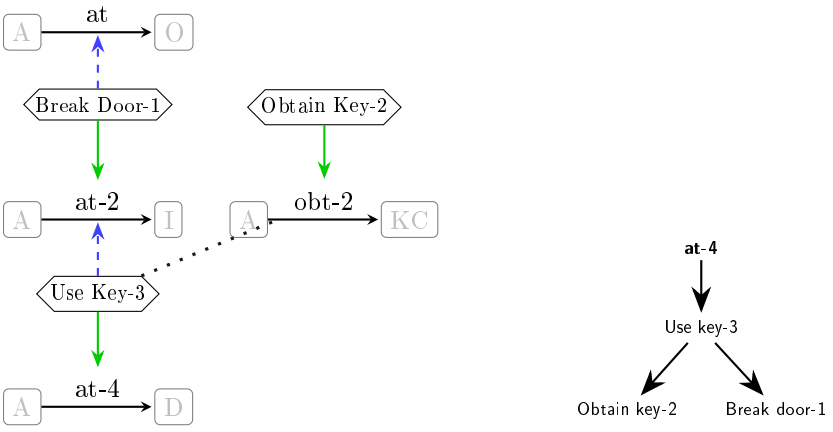
But while the gluing operation makes the exploration very efficient, using it is in many ways similar to making the monotonicity assumption during the exploration, where it is assumed that attacker capabilities cannot be 'lost' during the exploration. As argued in the introduction of this work however, this assumption is not a natural decision for every security-domain.

In the case of graph transformation, the fact that the gluing operation does not remove any elements from the graph makes it possible for combinations of graph items to appear in the graph that are not concurrent and therefore should not be able to exist at the same time, for example because one items creation depends on another item being consumed.

The result of the graph containing sets of items that are not concurrent is that it is then also possible to find matches for transformation rules that cannot really exist, as the items that make up the match cannot exist at the same time.

This means that the exploration, in addition to all the regular and possible attacks, might also find attacks that are not really possible, causing an overestimation of the possible attacks, a known side effect of the monotonicity assumption.

In order to avoid making the monotonicity assumption, the unfolding construction defines that only matches of which all graph elements do not depend on each other (i.e. can exist at the same time) should be explored in the unfolding, i.e. should be glued to the graph. Other matches should be ignored by not gluing them to the graph and thereby halting their exploration.



(a) Causal history at-4 (b) Configuration at-4

Figure 3.10: Example of the causal history and configuration of the at-4 graph element

The unfolding construction procedure therefore requires a filter to be implemented to determine if a match is valid or not, i.e. determine if its elements depend on each other or not.

3.2.1 Running Example

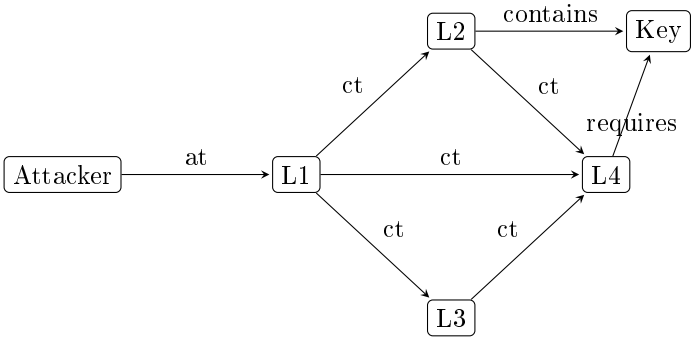


Figure 3.11: Startgraph of the example GTS that will produce invalid matches during its Unfolding

In order to give a better intuition into the issue of how elements that depend on each other can exist at the same time, and how these are filtered out later on, we will first introduce an example Graph Transformation System for which such elements appear during its unfolding.

The Setup The startgraph of this example GTS is depicted in figure 3.11. This input graph contains four locations, numbered L1 through L4. The attacker starts at location L1 and his goal is to move to location L4, which requires a key that can be found in location L2. Together with three transformation rules (Move, Obtain and *Use Key*), this is the setup the example GTS.

The main difference of this GTS compared to previous examples is that the key must be obtained from the model itself, not from 'an external' source. Therefore, intuitively, the attacker needs to move to L2 to obtain the key and then use this key to proceed to L4.

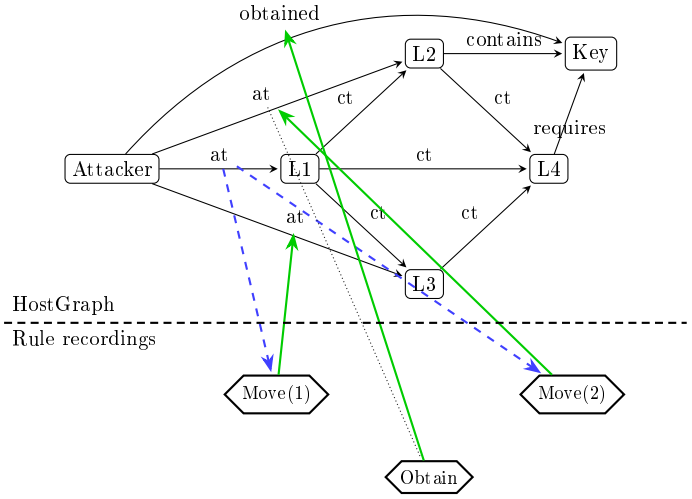


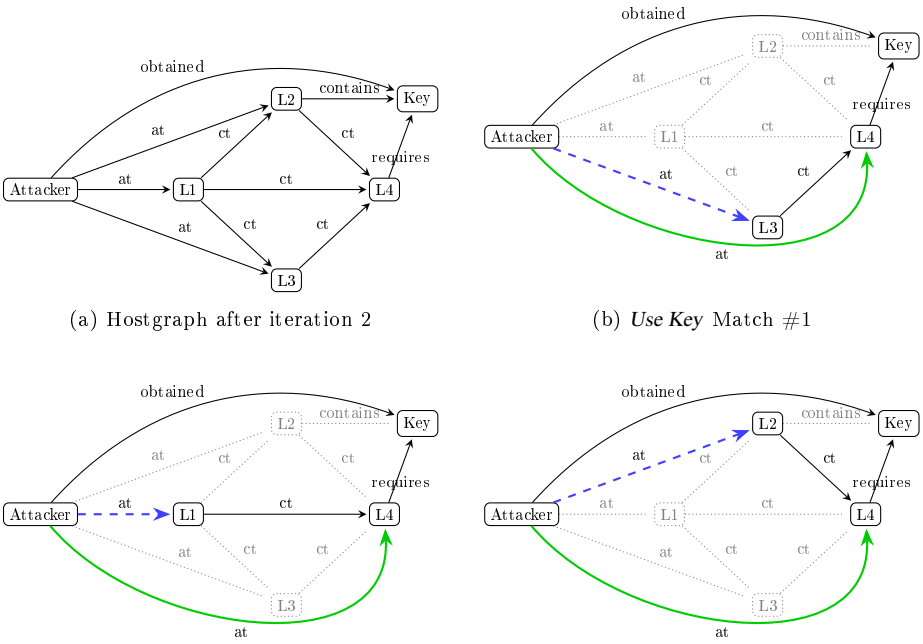
Figure 3.12: Unfolding of the example graph (figure 3.11) after the second iteration.

Unfolding construction The first two iterations of the unfolding work as expected. In the first iteration, the attacker can move to both location L2 and L3 with the Move transformation rule, but not to L4 as this requires the key to be obtained. In the second iteration, the attacker can obtain the key from L2.

This is all expected behavior and the resulting Unfolding at the end of the second iteration is shown in figure 3.12.

Third iteration It is however during the third iteration that the unwanted behavior occurs. At the start of the iteration, the unfolding construction searches for all possible matches of all transformation rules and will find 3 new matches in this iteration. These three new matches are shown in figure 3.13.

Of these three found matches however, only the third (figure 3.13d) is a valid matches. The other two are invalid matches, as there exists a combination of



(a) Hostgraph after iteration 2

(b) *Use Key* Match #1

(c) *Use Key* Match #2

(d) *Use Key* Match #3

Figure 3.13: The three rule matches found during the third iteration of the Unfolding construction, of which the first two are invalid.

elements in both of these matches that depend on each other. In order to obtain the key, the attacker has to move to location L2 and this move means that the attacker is no longer at location L1. Or in other words: the Move transformation rule for moving from L1 to L2 requires that the 'at' arrow towards L1 is removed.

Therefore, the 'obtained' arrow, which is part of all three matches (as an element of its reader set) depends on the 'at' arrow between the attacker and L1 being deleted. This 'obtained' arrow element thus is in conflict with the 'at' arrow to L1 still existing or even being used to move to L3. Only the third match is valid, as the 'obtained' arrow is not in conflict with the 'at' arrow to L2.

We refer to these conflicts as conflicts between the configurations (the set of rule applications they depend upon) of graph elements. The conflicts can also be seen in the unfolding visualisation 3.14 of when these three matches are glued to the graph.

As explained previously, the reason that these matches are found is because the gluing operation does not remove any elements. This fact, in combination with how the system is modeled, results in the attacker being modeled as though being in multiple locations at the same time.

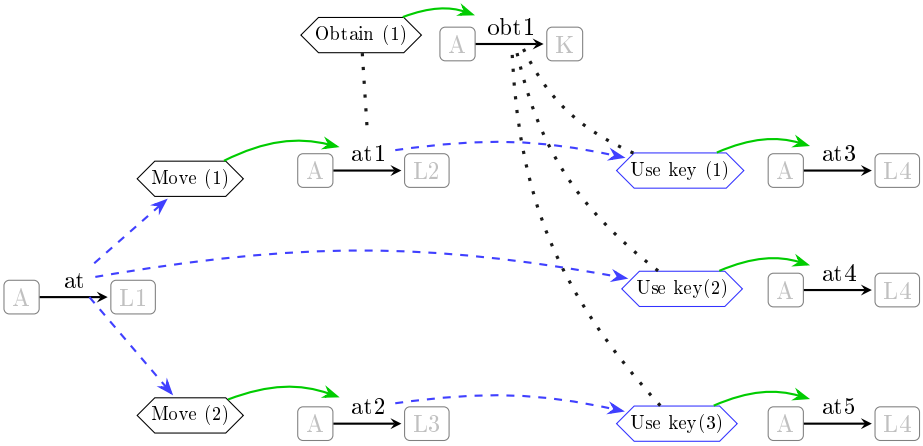


Figure 3.14: Unfolding visualisation of the example graph (figure 3.11) after gluing all three matches of the Use Key rule, containing two conflicts.

If these invalid matches are also glued to the graph, this would result in more attacks being found then are actually possible and thus resulting in the expected overestimation of attacks being found in the exploration. Therefore, we want to filter out these first two matches as invalid matches and not glue them to the startgraph.

3.2.2 Filtering

The task of the filter is to determine if a transformation rule match is reachable, i.e. if the elements of the match are concurrent and it is therefore possible to reproduce this match by applying the transformation rules it depends on instead of gluing them. If this is the case, the match describes an action that is possible according to the model, such as the attacker using the key to move from L2 to L4.

In order to understand how this can be determined, it is important to first understand that a certain graph element can only exists if the transformation rules that it depends upon have been applied to the startgraph. The 'obtained' arrow in our example GTS depends upon the Move to L2 and the Obtain transformation rule being applied to the graph for example. Therefore, in order to obtain all elements of a match, the rule applications in the configuration of each element have to be applied to the startgraph in the specified partial-order.

If there is an order in which all these rule applications can be applied to the startgraph that is in accordance with the partial-order of all configurations, the match is reproducible by rule application and therefore a reachable and valid match.

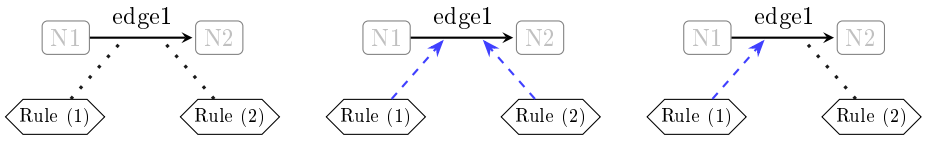
Therefore, the filter has to determine for each match if there exists an order to apply all its rule application dependencies to the startgraph.

Conflict types In order to understand the difficulty in determining if there exists such an order, it is important to first understand the different type of conflicts that can prevent such an order from existing.

All elements of the graph have a configuration that describes a sets of dependencies, i.e. rule applications that together created this element. Each of these transformation rule recordings has an effect, such as elements being consumed, read or created. Conflicts between two elements occur because there have transformation rule recordings with conflicting effects, e.g. they both have a rule recording that consumes the same element.

If the configuration of two elements does not interact with the same graph element, i.e. if the effects of all transformation rule recordings of the different sets do not mention the same graph element, such as attempting to consume the same graph element, there cannot be a conflict therefore there is a simple order in which all of these dependencies can be applied to the start graph, simply by adding each set independently. This is because the different sets have an internal order and applying the sets has no influence on the other set, as their effect does not focus on a shared graph element.

Therefore, there can only exist a conflict between the configuration of different elements if they interact with a common element.



(a) Double read interaction (b) Symmetric conflict (c) Asymmetric conflict

Figure 3.15: The three types of interaction with a shared element two configurations can have

If two sets of configurations read a common element, such as shown in figure 3.15a there is no conflict as the configurations can still be applied in either order.

If two sets of configurations however both consume a common element, as shown in figure 3.15b, the two configurations are in conflict, as there simply exists no order to apply both of these configurations that has no conflict.

Finally, if two sets of configurations interact with a common element, but the first consumes it while the second reads the element as shown in figure 3.15c, there might be a conflict. This depends on whether there exists an additional conflict of such a type that, when composed with the first one, results in an asymmetric conflict.

A set of asymmetric conflicts can be composed into a symmetrical conflict, see 3.16. It is this fact, that a set asymmetric conflicts can be composed into a symmetrical conflict, that makes it hard to determine if the configurations of the elements of a match have an order or if a composed conflict exists.

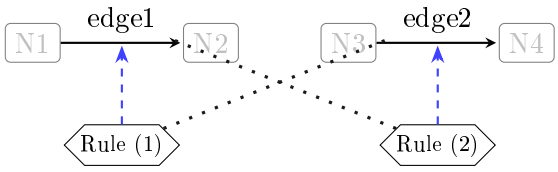


Figure 3.16: Example of two asymmetric conflicts being composed to a symmetric conflict

The algorithm A conceptually simple method to determine if there exists an order is to simply identify such an order before adding a match tot he graph. The issue lies in finding such an order in an efficient manner however, as the filter will have to be applied to each possible rule match.

Baldan *et al.* [19] have proposed an approach for efficiently identifying if there exists a possible order for events of Petri nets with read arcs. As such so called *contextual Petri nets* are similar to graph transformations systems their approach can be translated to this domain.

At the basis of this approach lies the idea that each rule application match can have a set of possible explicit histories based on whether or not the elements consumed in the history of a match have been read by other rules before they were consumed or not.

The approach is based on annotating all elements of the host graph during the unfolding with all the possible reading and creating histories of that element, which can be done in an iterative manner, and then using this to identify new possible histories for matches of transformation rules and adding these new histories to graph elements.

The approach then works as follows. When the approach identifies a possible history for a match of a rule, the match is added to the graph and the history of this match is stored in the elements the rule application reads and creates as respectively reading and creation histories of the match. Following iterations can use the histories stored in the graph elements to identify new histories of other matches, which are once again stored in graph elements to identify even more new histories.

In this fashion, all possible histories of a each match are iteratively constructed during the unfolding. This results in a low complex approach to identify whether each match has a valid history.

The reason for storing all histories explicitly is that this makes it possible to determine if two histories, i.e. the history of two elements of a match, contains conflict or not. Even composed symmetric conflicts can be identified in this manner

This works approach has implemented Baldan *et al.* proposed lazy algorithm for identifying possible histories. The specific details of the approach, including its proof, require are more formal introduction which can be found in the referenced work.

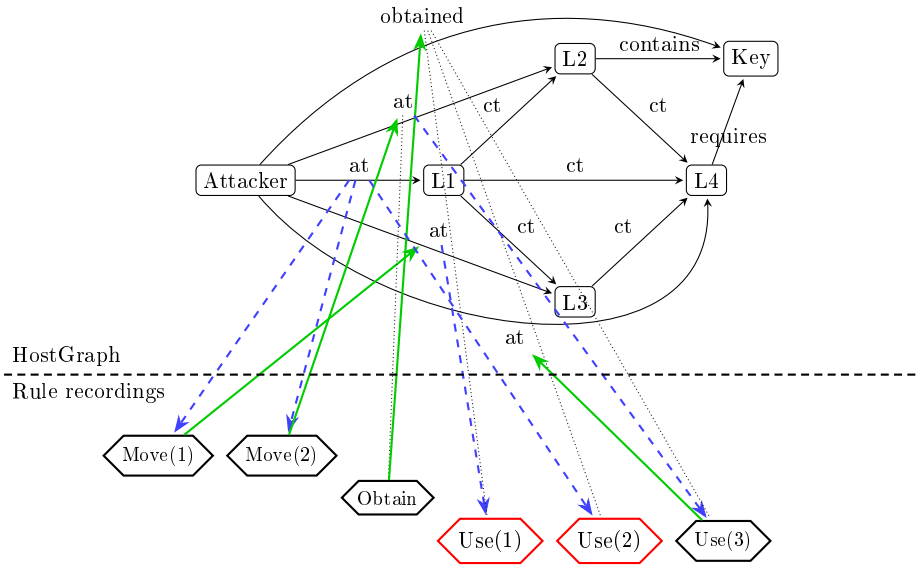


Figure 3.17: Unfolding of the example graph (figure 3.11) after the third iteration, including the filtering of invalid matches, shown as red (not-glued) rule recordings.

The result In order to demonstrate the result of applying the proposed filter algorithm to all matches found in the unfolding construction, and then only gluing the valid matches to the graph and discarding the other matches, the unfolding of the example GTS was extended with a third iteration. The resulting unfolding is shown in figure 3.17.

The two matches found to be invalid are recorded in the unfolding as invalid matches (so that it is remembered that they are invalid when they are found in the following iterations), but they are not glued to the graph and therefore not explored.

Only the 'Use(3)' transformation rule match is valid and therefore glued to the graph, resulting in only one attack route to L4 being found, as was expected.

3.3 Limitations

The previous sections describe the foundation of the unfolding construction approach. This approach does however not (yet) support all graph transformation features. This section gives an overview of the the current limitations of the described approach to highlight when the approach can be used and gives insights into how these limitations can be resolved in future work.

3.3.1 Consuming and creating nodes

The current unfolding implementation only supports transformation rules with nodes as reading elements, not nodes that get created or consumed. This feature is currently only implemented for edges. Support for the creation and removal of nodes in the transformation rules was not implemented due to a combination of time limitations and the fact that the case study model did not require this feature. This is however purely a implementation issue: The theory behind the approach, as discussed in the previous sections, works for both the edges and nodes of a graph in a similar manner.

3.3.2 Negative application conditions

The most important feature that is currently not yet supported is the possibility of expressing negative application conditions. Supporting this feature (greatly) increases the specification possibilities of the Graph Transformation paradigm and it is therefore an important feature.

The general unfolding construction introduced so far is however not capable of handling this class of transformation rules that contain negative application conditions. Because of the gluing operation, the entire exploration approach is based on not removing any elements from the graph during its exploration. The result of this is that when a GTS that contains rules with negative application conditions is explored using the gluing operation, not all matches of all rules are always found during the exploration, as there might exist elements in the graph that prevent this specific class of rules from having an applicable match in the graph.

Not finding all valid rule application matches is a severe drawback in the case of finding attacks, as the approach aims to satisfy the *exhaustive* and *succinct* property.

On the other hand, the gluing operation is a cornerstone of the unfolding construction approach. Altering its working by removing elements from graph in order to find matches for this type of transformation rule would undermine the entire exploration approach.

Baldan *et al.* [33] have encountered the same drawback when they used the unfolding constructing approach to generate test cases for code generators. They have therefore proposed an extension to the unfolding construction approach so that it support transformation rules with negative application conditions while keeping the efficient exploration approach. Their proposed extension to the unfolding construction is discussed in this subsection.

3.3.2.1 Unfolding construction extension

In order to understand the extension, it is important to realize that this new class of transformation rules, those that contain negative application conditions, introduce an additional set of elements when compared to the three previously

introduced (Reader, Consuming and Creator sets), namely the set of inhibitor elements.

The goal of the extension is to have all recordings of transformation rules refer to graph elements that prevent them from being applicable, similar to how the recordings already refer to the elements that they consume.

But as mentioned before, during the exploration of a GTS using the gluing operation, not a single element will ever be removed from the graph. Therefore, depending on the specific transformation rules, there is a chance that not all matches of the transformation rules with negative application conditions will be found.

In order guarantee that all matches of all transformation rules are found during the unfolding, while still keeping the efficient exploration of the gluing operation and thus not removing any elements from the graph, the extension proposes the following two steps:

1. Of all transformation rules with a negative application condition, the elements of the rule that describe this inhibitor of its applicability are removed for the exploration.

   In this way, during the exploration with the gluing operation, all matches of this modified transformation rule are found and recorded. (All matches of this modified transformation rule include all matches of the unmodified rule.)

2. The second step comes after the entire unfolding has been constructed with the regular approach and modified rules. In this unfolding, every recording of the modified transformation rules is considered and, for any falsifying element, a negative application condition reference to this edge is added to the rule recordings inhibitor elements set (just as the other three types of elements are already added to their respective set in the rule recording).

This second step then produces the final unfolding. From this final unfolding, it can then be determined if all matches of the modified transformation rules are still valid matches and what their additional dependencies are, such as transformation rules that remove their falsifying edges.

3.3.2.2 Running example

Let us demonstrate this extension with an example.

The setup: The setup is a simple start graph (see figure 3.18) with just three locations and an attacker that is at location L1. The example GTS has two transformation rules, a (special) 'Move' transformation rule that specifies that an attacker can only move to a location if this does not require a key, and a 'Break Lock' transformation rule that can remove the need for a Key.

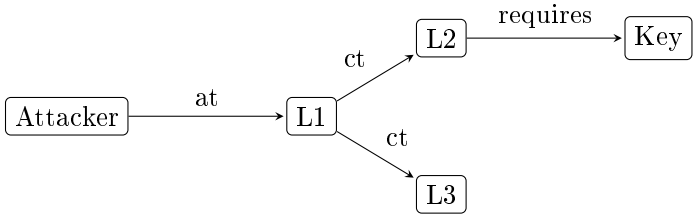


Figure 3.18: Startgraph of the example GTS that contains a rule with a negative application condition

The Unfolding Construction The first step of the extended unfolding construction is to obtain a modified version of each transformation rule that contains negative application conditions, namely a version of this rule where all such inhibitor elements have been removed.

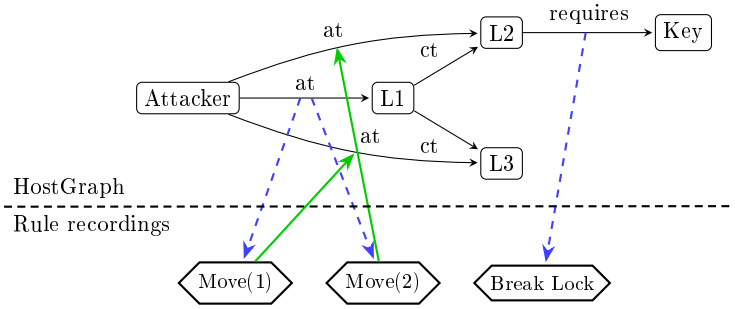


Figure 3.19: Regular unfolding of the example GTS's startgraph (figure 3.18)

Using these modified versions of the transformation rules, the unfolding construction then continues as regular, finding all matches for all transformation rules in each iteration, and gluing all valid matches to the graph.

The result of the regular unfolding can be seen in figure 3.19. The unfolding has found two applications for the modified move version, including one to move the Attacker to location L2, as the modified Move rule is not bothered by the existence of the 'requires' arrow of L2 (unlike the original transformation rule). In addition, the Break Lock transformation rule has one match where it removes the 'requires' arrow from the graph.

The second step of the extended unfolding construction obtains the unfolding constructed by the regular exploration and modifies it by adding negative application condition references to all modified rule recordings for all falsifying edges of that rule application. The resulting and final unfolding can be seen in figure 3.20.

From this unfolding, it can be seen that the rule recording for 'Move(1)' now has a dependency in the form of the 'Break Lock' transformation rule.

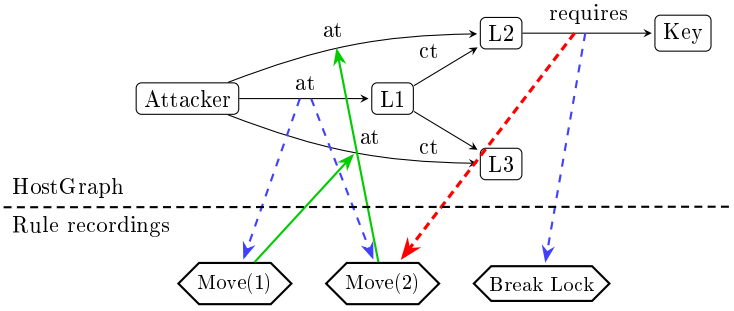


Figure 3.20: Final unfolding of the example GTS's startgraph (figure 3.18, after the inhibitor elements have been added to the rule recordings.)

Only when this second transformation rule has been applied can the Move(1) transformation rule be applied.

3.3.2.3 Discussion

The proposed extension is however not yet implemented because it introduces non-determinism to the unfolding which increases it's complexity. The reason for this is that there might be multiple elements in the graph that would prevent a recorded rule from being applicable. Simply adding this information to the unfolding would remove the single branching structure property.

While they have demonstrated that this extension can be used, Baldan *et al.* are not clear on how this foundational property of the unfolding can be upholded. Therefore, it is left to future work to determine how this extension can be added to the unfolding construction to add support for negative application conditions.

3.3.3 Cycles reduce performance

A GTS can be modeled in such a way that it contains a cycle, i.e. a set of transformation rules that, after applying all of them, returns the graph to the same state as before they were applied.

Another limitation of the current Unfolding Construction approach is that it does not realize if it is exploring a cycle and it will therefore keep exploring this cycle indefinitely.

Therefore, if the unfolding construction finds such a cycle, the construction approach is in principle infinite and would continue to increase in size for an arbitrarily long time.

3.3.3.1 Running Example

We introduce a small example GTS that contains such a loop to show how the unfolding construction will handle this cycle.

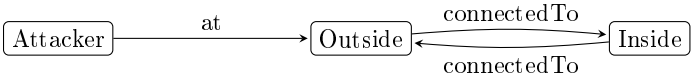


Figure 3.21: Startgraph of the example GTS that contains a cycle.

The setup is a simple startgraph (see figure 3.21) that contains two locations. The main difference with previous examples is that the two locations are connected in both ways, meaning that the attacker can go back and forth.

The regular unfolding construction exploration will find it can move to the second location in the first iteration and then move back to the first location in the second iteration. Because it then creates a new 'at' arrow to the first location, the third iteration will find an additional match to move to the second location, also adding a second 'at' arrow to the second location, which in turn can be used to move back again to the first.

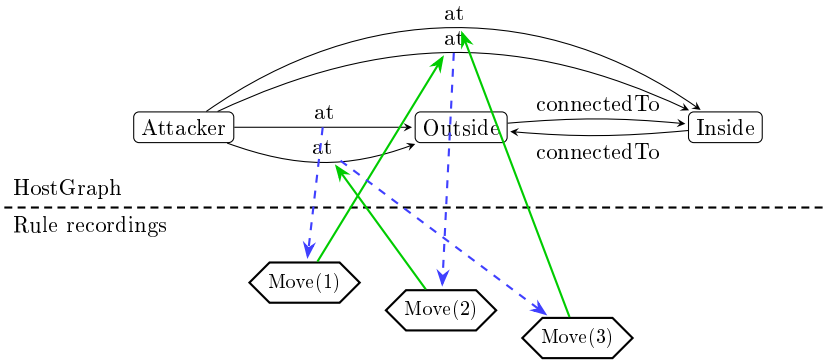


Figure 3.22: Unfolding of the example GTS's startgraph (figure 3.18) after the third iteration, showing how a loop can be explored indefinitely.

The unfolding after iteration three that can be seen in figure 3.22 demonstrates this point. Because the gluing operation keeps adding annotated elements to the graph, new matches get created which keeps the exploration going indefinitely.

3.3.3.2 Solutions

Depth restriction Baldan *et al.* have proposed a simple solution to this issue of infinite exploration in the form of a depth restriction. A depth restriction is based on halting the exploration after a certain amount of iterations have been concluded and thereby guaranteeing that the approach finishes.

The downside of this approach is that it introduces a significant performance burden when an input model contains a cycle. Until the depth restriction is reached, the approach will keep exploring the cycle and the new histories it produces. While these additions to the unfolding can easily be ignored after it's

constructed, the exploration will perform unnecessary exploration which will decrease its performance.

In addition: choosing a sensible depth restriction is also non-trivial. On the one hand you want to guarantee that even the longest possible attacks are found, but choosing a large depth restriction will likely produce a lot of unnecessary exploration of the cycle(s).

Cycle detection A more elegant solution would be to detect cycles during the exploration. This detection could be done for each new match found during the exploration and then determining if applying this match would return the state of the graph to one already explored. If this is the case, the exploration is halted by not gluing this transformation rule to the graph.

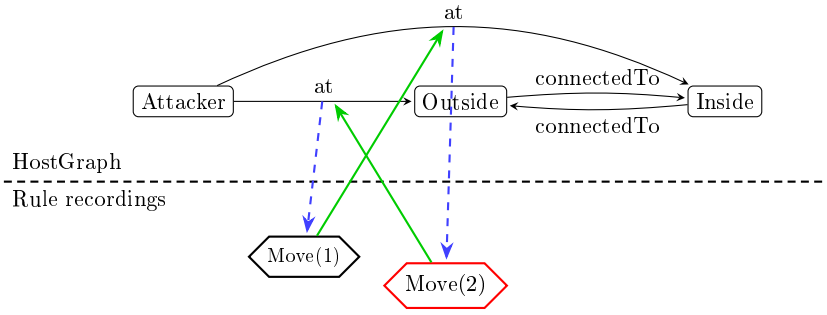


Figure 3.23: Example of halting the unfolding after a cycle has been detected.

Figure 3.23 shows how such a detection might work. The unfolding construction finds that the second Move rule applications returns the graph to a previously visited state and decides to halt the exploration.

There are however a few difficulties in detecting a cycle. While the cycle in the example is easy to detect, it can be much harder to detect cycles in larger GTSs. These cycles are not only conceptually difficult to find, they are also computationally difficult to compute.

In addition to these difficulties there is another issue. If the extended unfolding construction approach that supports negative application conditions is used, it is even more difficult to detect if the unfolding is really exploring a cycle, as exploration using the unmodified transformation rules might not result in a cycle.

Hybrid solution We believe the preferred solution would be a hybrid of these two solutions. On the one hand, the cycle detection prevents unnecessary exploration of the GTS, which requires computation resources and produces uninteresting rule recordings for the unfolding, but it hard to detect all cycles and it does not guarantee a completion in combination with negative application conditions.

On the other hand, while the depth restriction guarantees a completion of the unfolding, only using a depth restriction will result in computational resources being used to investigate cycles and produce uninteresting rule recordings.

Therefore, a hybrid solution could be based on detecting simple cycles, in order to minimize the computational resources used for the exploration of cycles, but at the same time guarantee a completion of the unfolding construction.

Discussion Currently the depth restriction solution is implemented to guarantee that the unfolding construction finishes.

This can easily be extended with methods that identify 'simple' cycles, which will already reduce the unnecessary exploration, but a method to identify all possible cycles is more complex and is left for future work.

Chapter 4

Part 2: Constructing the Attack Tree using the unfolding

All models are wrong, but some are useful.

George E. P. Box

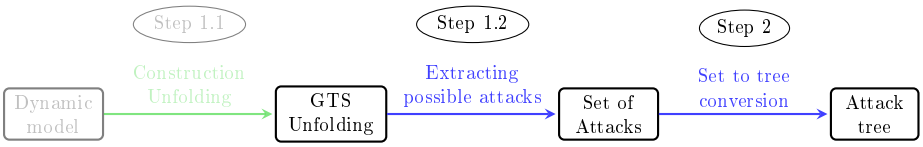


Figure 4.1: This chapter discusses step 1.2 and step 2, extracting the attacks from the Unfolding and converting them into an attack tree.

Given the procedure to construct the unfoldidng of a GTS as was described in the previous chapter, there are still two steps to obtain the corresponding attack tree as shown in figure 4.1. This chapter discusses how these two tasks can be performed.

4.1 Extracting attacks from the Unfolding

Given a completed unfolding that represents the exploration of a GTS, the next step is to identify all possible attacks and extract them from the unfolding. This section will demonstrate how this is done.

Identifying attacks The first challenge is to identify all the attacks in the unfolding. This can be done using a (goal) condition rule. Such a condition rule

defines when the attacker has achieved his goal and therefore each match of the rule in the graph represents a possible attack that can be found in the model.

Therefore, the first step is to find all matches of a goal condition rule in the resulting graph at the end of the unfolding. In order to extract these attacks from the unfolding, each match of the condition rule is also recorded in the unfolding, so that the unique set of dependencies of the goal rule match (which represent the steps of the attack) can be determined.

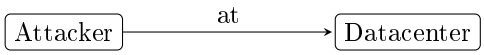


Figure 4.2: A simple goal condition rule that is used to find attacks in the unfolding.

This procedure is demonstrated using the example unfolding constructed in the larger example subsection of previous chapter, see figure section 3.1.4. The first step is to add a goal condition rule to the GTS, in this case the simple condition rule shown in figure 4.2 suffices. This condition rule states that the attackers goal is to reach the data-center.

The next step is to find all matches of this goal condition rule (that represent the attacks found in the exploration) and to record these matches in the unfolding.

The result of this step is shown in figure 4.3. The example unfolding of the previous chapter is extended with four recordings, one for each of the four matches of the goal condition rule found in the host-graph of the unfolding.

Extracting attacks After identifying all attacks by recording all matches of the goal condition rule, the actual attacks can be extracted from the unfolding by finding the (unique) set of dependencies of each goal match. Each dependency of the recorded match represents an attack step of the attack, with the set of attack steps (i.e. all dependencies) representing the attack itself.

The resulting set of dependencies of each attack can be extracted from the unfolding in what we refer to as a dependency structure.

This extracting procedure performed on the unfolding of figure 4.3 results in the four dependency structures shown in figure 4.4, where the arrows denote the dependencies between rule recordings of the unfolding.

For example, the dependency structure of the first attack, shown in figure 4.4a simply describes that the goal match requires the action Break Door (2) to be performed, while that action depends on Break Door (1) being performed.

The second example, shown in figure 4.4b is a bit less straightforward. The goal here depends on the Use Key (2) rule application. This rule application has however to dependencies, both Obtain Key (2) and Break Door (1).

In a similar manner, the dependencies of the other two attacks can be interpreted.

All of this dependency information between rule applications can be retrieved from the unfolding by looking at the causal history of a graph element, as was

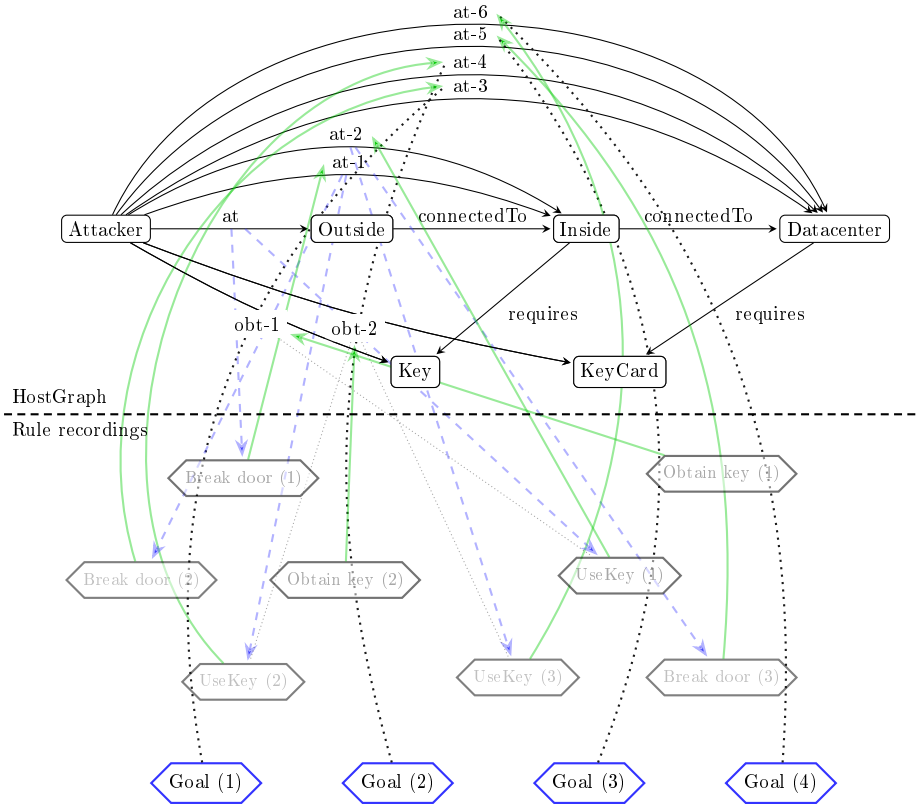


Figure 4.3: Unfolding including all matches of the goal condition rule.

shown in section 3.1.4.

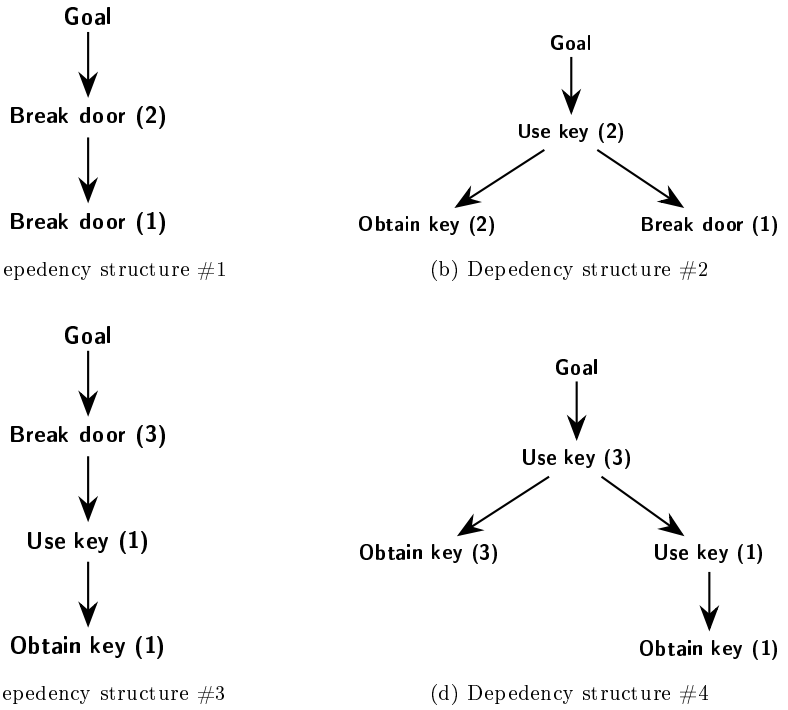
4.2 Converting a set of attacks into a tree

Given a set of attacks, the next step is to convert them into a single (attack)tree representation.

In principle there is an attack tree equivalent for each dependency structure obtained from the unfolding, depending on the attack tree formalism variant that is chosen.

Traditional Attack Tree formalism The default attack tree formalism only supports two type of gates: AND gates, of which all children must be true, and OR gates, of which at least one child must be tree. The main interesting point is that the ordering of the children of an AND gate has no semantic meaning.

Figure 4.5 shows the traditional attack tree equivalent of the attack described



(a) Depedency structure #1

(b) Depedency structure #2

(c) Depedency structure #3

(d) Depedency structure #4

Figure 4.4: All four dependency structures, each representing an attack, that can be extracted from the example unfolding.

by dependency structure #2. This attack consists of three steps: Break door 1, Obtain key 2 and Use key 2. All three attack steps must be completed before the attack is completed, which is demonstrated by the figure.

However, while the dependency structure gave additional information about the order in which these attack steps must be performed, such as 'Obtain key' before 'Use Key', this is lost in this traditional attack tree because of the semantics of the AND gate. An attack tree where Use key 2 would be the first node of the AND gate is considered to be equivalent to the one shown above.

Even though in this case, the order between Obtain key 2 and Use key 2 is specified in the order of the leaf nodes, this information cannot be used, as it is not always possible to specify order in this way, as can be seen by the relation between Break Door 1 and Obtain key 2. There is no order between those nodes, the only requirement is that both need to happen before Use Key 2 can be performed.

Extended Attack Tree formalism with SQAND There are however extensions to the default attack tree formalism (for example *Improved attack trees* [41]) that supports a so called sequential AND gate (SQAND) which specifies that the order of the children has a semantic value and thus can be used to

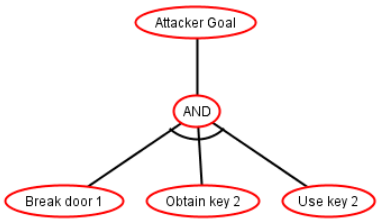


Figure 4.5: Traditional attack tree formalism equivalent of dependency structure # 2

specify the order in which actions should be performed.

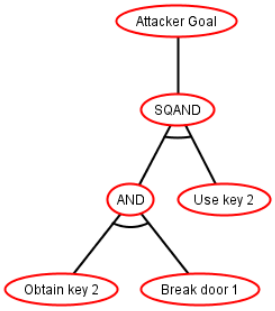


Figure 4.6: Attack tree formalism extended with SQAND equivalent of dependency structure # 2

Figure 4.6 shows the attack tree with SQAND gates equivalent to the attack described by dependency structure #2. The attack tree specifies that both Obtain key 2 and Break door 1 need to happen before Use key 2 can be performed, but does not specify an order in which the first two actions have to be performed.

Single tree composition Given an AT for each individual attack (independently of the formalism used), it is easy to construct a single attack tree for all attacks, simply by defining it as an OR composition of all attacks.

En example of such a single attack tree, for all four attacks and their extended AT equivalent, can be seen in figure 4.7. The attacker goal node, modeled as an OR gate, is achieved if any of these attacks succeeds.

4.3 Towards a compact tree representation

The simplest method for constructing a tree representation of a set of attacks, as shown in the previous section, is to construct the attack tree for each individual

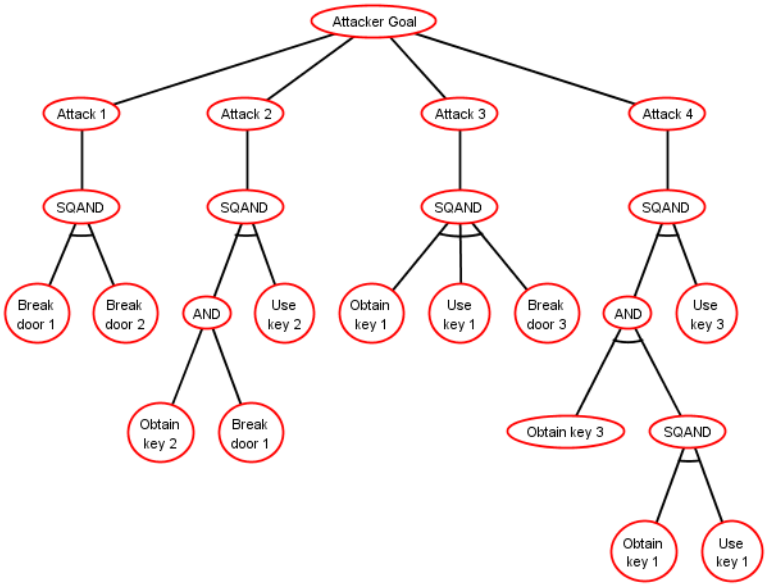


Figure 4.7: All attacks combined into a single attack tree using an OR gate

attack and develop a composite attack tree that represents all of these attacks separately as options to reach the goal.

However, the downside of this method is that it often results in large attack trees that are hard to analyse for both humans and analysis tools. Therefore, the preferred method is to determine if a more compact attack tree, i.e. a more compact description of all possible attacks, can be constructed. This section discusses how a more compact attack tree representation can be obtained.

4.3.1 Intuition

An important point to understand is that while each attack consists of a unique set of attack steps, the attack steps themselves are not unique for each attack, in fact, many attack steps are often part of multiple attacks. In the default attack tree conversion method, each attack step is specified in the tree once for each attack it occurs in, resulting in multiple specifications of the same attack step in the complete tree. A compact tree representation attempts to exploit this feature of shared attack steps by reusing its definition in the representation of multiple attacks. This reduces the amount of (duplicate) attack steps specified in the tree and thereby reduces the size of the resulting attack tree.

The first two attacks described in figure 4.4 are a good example of two attacks that share an attack step. Both attacks are based on first using the breaking door attack step to get to the Inside location, but then use a different method to reach the Datacenter location. In the default attack tree representation of

this set of attacks, shown in figure 4.7, it can be seen that each attack has its own sub-tree, and each sub-tree defines the break door attack step once.

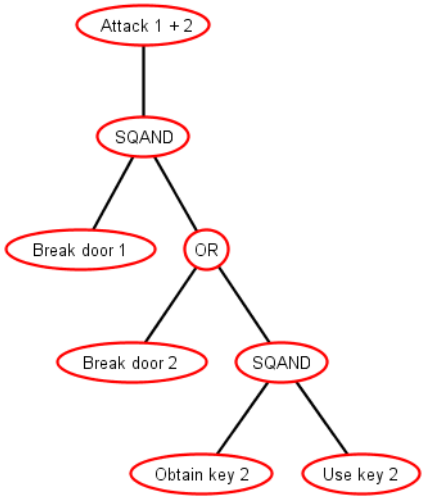


Figure 4.8: Example of how to attacks can be merged into a mare compact representation

The two sub-trees representing these two attacks can be combined into a single sub-tree that defines the initial break door attack step only once. An example of such a combined representation can be seen in figure 4.8. This tree representation still uses the SQAND gate to specify that the break door step needs to be performed first, but combines this with an OR gate to specify that there are different ways to continue the attack.

In order to understand why the definition of an attack step can be reused for multiple attacks, it is important to realize that attack steps are essentially stateless, meaning that the effect of an individual attack step is always the same (and is not influenced by the other steps of the attack).

The definition of an attack step in an attack tree only represents its effect, e.g. using a key to open a door, which is independent of the other actions of the attack. Therefore if another attack contains the same attack step, its effect is also independent of the other attack steps, and thus the two definitions of the attack step in an attack tree are equivalent and the first definition can be reused for the second attack. This reuse also has no influence of the attack itself, as attacks are represented as sets of stateless steps each with its own effect. As long as the combined effect of all individual steps remains the same, the attack remains the same.

4.3.2 Obtaining the general attack step of each rule recording

An interesting effect of the unfolding construction is that it records many very specific attack steps, namely it records each application possibility of each production rule for each unique history of that application possibility. This is done in order to obtain a single branching structure of the behavior of a GTS.

However, for the purpose of an attack tree, these attack steps are often too specific. As mentioned before attack steps are stateless, and thus the history of an attack step has no influence on its effect. This results in the fact that the constructed unfolding often contains multiple recordings of the same general attack step, i.e. multiple attack steps that have the same effect, because each recording is annotated with its unique history.

As also mentioned previously, in the compact representation of a set of attacks, the goal is to reuse the definition of general attack steps that are shared between attacks. However, these multiple recordings of the same general attack step make it difficult to determine if two attacks share general attack steps, i.e. if two attacks have attack steps with the same effect. Therefore, in order to merge attacks extracted from the unfolding, the recorded specific attack steps first need to be converted to their general attack step equivalent, i.e. the attack step without its history attached to it.

The recorded production rule applications in the unfolding are currently annotated with all the elements of their match in the graph. Because each of these elements is in turn annotated with the recording of the production rule that created it, this annotation is represents its unique history.

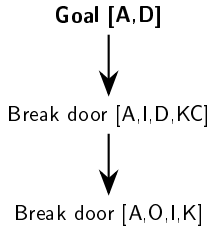
It is proposed to convert these recordings into their general attack step equivalent by removing the history aspect from the annotation. This is done by replacing the existing annotation with an annotation that describes the unannotated elements of its match, instead of the annotated elements. This annotation will result in the fact that attack steps with the same effect, i.e. production rule recordings of the same rule with a the same unannotated match, get the same annotation and are therefore easily recognizable as equivalent attack steps, while other recordings of the same production rule, i.e. those with other matches, still get different annotations.

A good example of this issue of multiple recordings of the same general attack step can be seen in attack 1 and 3 from figure 4.4. Both use the same general attack step to go from the Inside Location to the Datacenter, namely the break door rule recording. However, because each attack has a different history of getting inside, there are two different recordings of the break door production rule, namely Break door (2) and Break door (3). These two production rule recordings operate on a similar match, only the annotation of one of the elements of both their matches is different (as can be seen in the unfolding in figure 4.3), and they therefore represent the same effect.

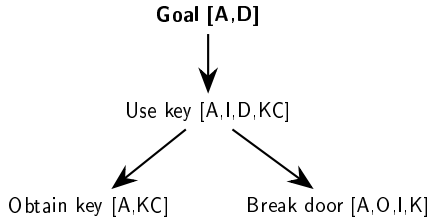
In order to convert these two rule recordings into their general attack step equivalent, the annotation of the recording is altered to an annotation that

does not include the annotations of the elements of the match, only the general elements. In the case of the two break door rule recordings, both are annotated with the following: [Attacker, Inside, Datacenter, KeyCard, Attacker-at-Inside, Inside-ConnectedTo-Datacenter, Datacenter-requires-KeyCard].

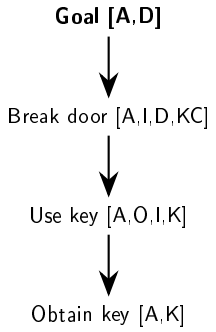
In the same manner all other recordings are converted and annotated the general items of their match, resulting in all recordings that represent the same general attack step get the same recording, making it easy to compare them.



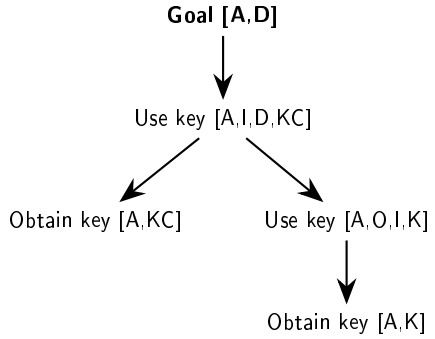
(a) Annotated dependency structure #1



(b) Annotated dependency structure #2



(c) Annotated dependency structure #3



(d) Annotated dependency structure #4

Figure 4.9: All four annotated dependency structures that each represent a set of general attack steps

The result of annotating all rule applications of the unfolding in this manner results in the four attacks shown in figure 4.9. For convenience, all rule recordings are only annotated with the first character of all node elements involved in the match, which is sufficient to distinguish between all matches of this example. From the examples, it is now much easier to determine what attack steps are shared between the different attacks. If their string representation is the same, they represent the same general attack step and thus the same effect.

4.3.3 Proposed merging startegy for SQAND support

The following subsection discusses a proposed merging approach for attacks that is based on maintaining support for the SQAND gate and therefore keeping

the dependency information of attack steps in tact during the merging. In order to maintain this information, the mering approach focuses on merging the dependency structures themselves into a combined dependency structure that maintains all partial-order information between attack steps.

Dependency structures can be merged by identifying their shared dependency chains, i.e. chains of attack steps that depend on each other, instead of identifying the individual shared attack steps. These identified dependency chains need to start from the top of each dependency structure in order to guarantee the chain is not a dependency of other attack steps (which is required to reduce the complexity of the merging).

Once the shared dependency chains are identified, the dependency structures can be merged based on this chain, meaning that the resulting combined dependency structure only defines the shared chain once. As the combined dependency structure can have different dependencies itself in the individual attacks, the resulting dependency structure shows each of these sets of dependencies as a possible dependency set of the shared chain.

The combined dependency structure can therefore be seen as a compact representation of a set of individual dependency structures and has an equivalent attack tree which represents the compact tree representation of the set of attacks.

Using the conversion of all attack steps to their generic variant, it is easy to find these shared dependency chains of two attacks. Attack 1 and 3 from figure 4.9 share a simple dependency chain, namely that the Goal[A,D] step depends on Break door[A,I,D,KC] attack step.

This shared dependency chain has however different sets of dependencies in the specific attacks, these different sets of dependencies can be seen as different 'routes' to reach the dependency chain of attack steps.

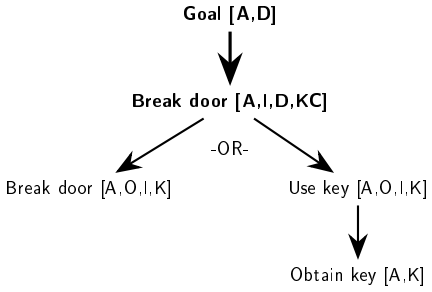


Figure 4.10: Example of merging two dependency structures (show in figure 4.9a and 4.9c) using their shared dependency chain.

These two example dependency structures can be merged using the identified shared dependency chain and an example of this can be seen in figure 4.10. Here the dependency chain is denoted using bold text. The different dependencies

of the shared dependency chain are simply depicted as different sets of dependencies of the chain by using an OR, meaning they describe different routes to satisfy the requirements of the dependency chain of attack steps.

While this simple example, the shared dependency chain only had one direct dependency in each attack, it can also have multiple direct dependencies, as is the case with the second and fourth attack shown in figure 4.9. In this case, the dependency chain can have multiple sets of direct dependencies.

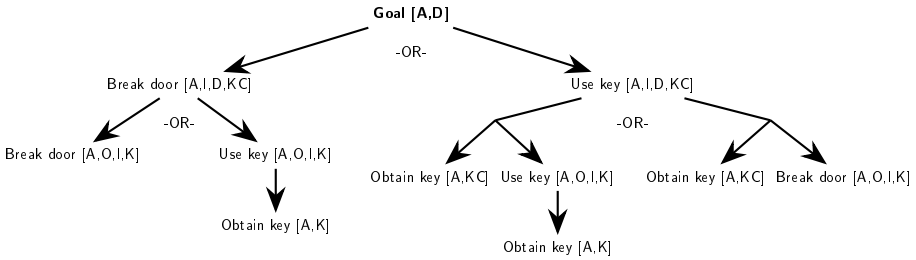


Figure 4.11: Merged dependency structure composing all four attacks

The dependency structure of all four attacks can be seen in figure 4.11. From this figure it can be seen that the second and fourth attack share a dependency chain, namely goal[A,D] depends on Use key[A,I,D,KC], and this dependency chain has two different sets of direct dependencies.

In addition, the figure demonstrates that the first and third attack do not have a common dependency chain with the second and third attack, they only share the same top step of reaching the goal.

The compact dependency structure of all individual attacks can then be converted into its attack tree equivalent, which is shown in figure 4.12.

Limitations The proposed merging strategy does not guarantee the optimal compact representation of a set of attacks, in-fact in most cases it will result in the most compact representation possible. The reason for proposing this specific strategy however is because it has a low computational complexity (bounded linear to the amount of attacks) and we argue that it produces human readable trees.

4.3.4 Other improvements/optimizations

In addition to merging similar attacks, there are three additional options to improve the resulting attack tree.

The first type of improvement is to detect shared attack steps in the sets of direct dependencies of a shared dependency chain. The shared dependency chain of attack two and four, as shown in figure 4.11, has two sets of direct dependencies with both these sets containing the 'Obtain key[A,KC]' attack

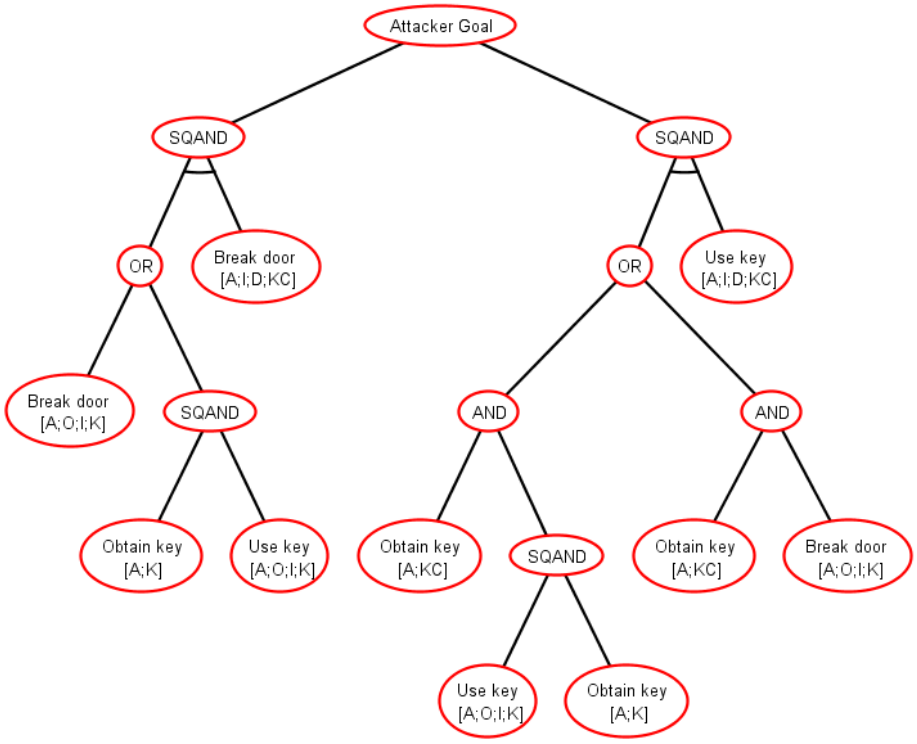


Figure 4.12: Attack tree corresponding to merged dependency structure

step. Therefore, such a shared direct dependency can be defined only once in the resulting attack tree.

In addition, the information contained in the unfolding also provides us with the information two determine if two set of dependencies of a shared dependency chain are mutually exclusive or not, e.g. if both sets of dependencies can be applied to the same graph. If two sets are mutually exclusive, these two sets can be modeled by an XOR gate in the attack tree, meaning that only one of the options of the gate can be true, or only one of the sets of dependencies can be applied at the same time. Using such XOR gates in the attack tree can result in reduced computational complexity in the analysis of the attack tree.

Finally, using a different attack tree formalism, it is possible to detect shared sub-trees in the resulting attack tree and transform the tree into a DAG structure where such shared sub-trees are only defined once. An example of a shared sub-tree can be seen in figure 4.12. Both the left and the right side of the tree contain a SQAND gate with obtain key and use key as its children. Larger trees are likely to have larger shared sub-trees defined multiple times in the tree. By defining such a sub-tree only once, the analysis methods only have to perform the calculations for the sub-tree once.

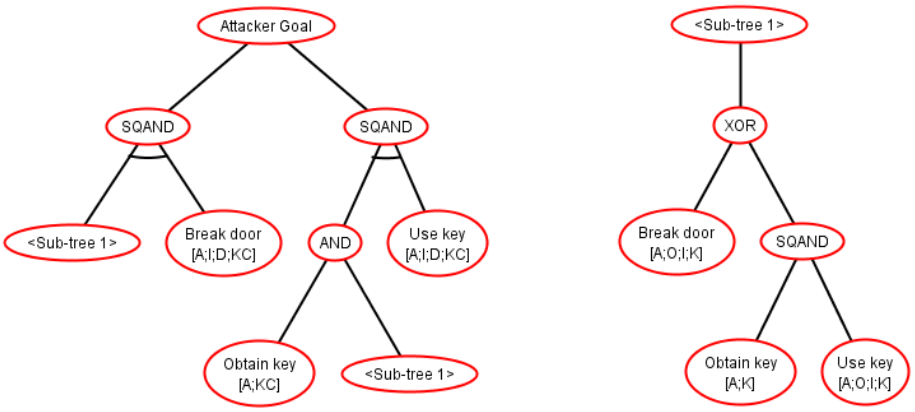


Figure 4.13: Optimized Attack tree of figure 4.12 with XOR, common direct dependencies detection and shared sub-tree support.

All these tree options to improve the tree are optional steps that will require additional computation to perform them. But all three have the potential to result in a reduced size/reduced computational complexity of the generated tree and therefore improve the (automated) analysis of this tree.

An example of all these three type of improvements to the attack tree shown in figure figure 4.12 can be seen in figure 4.13. On the right of this figure, the shared sub-tree is separately shown to indicate that it can be placed on two anchors in the main tree (shown on the left).

Chapter 5

Evaluation

Assumption is the mother of all screw-ups

Mr. Eugene Lewis Fordsworthe

The previous chapters have introduced the components of a complete attack tree generation approach. The goal of this chapter is to evaluate the individual components and and the complete approach in order to provide answers to this project's research questions.

The evaluation is divided into two main parts, a quantitative evaluation and a qualitative evaluation. The quantitative evaluation focuses on evaluating the scalability of the proposed approach to identify all attacks from a model. The qualitative evaluation focuses on several aspects, including graph transformations as input language, the usability of the output trees and the genericity of the complete approach.

This chapter starts with an overview of the implementation of this project's automated attack tree generation approach that is used for the evaluation.

5.1 Implementation details

The goal of this section is to give an high level overview of the implementation and the decisions made during the implementation process. The actual implementation can be found on the projects GitHub repository[1].

5.1.1 Implemented process overview

The implemented process follows the same steps as described in Chapter 3 & 4. The approach takes a GTS as input model. This model is unfolded iteratively until either it has been completely unfolded or a depth threshold has been achieved.

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

From the unfolding a set of attacks can be retrieved. Each attack is described through a dependency structure (that represents the partial-order of the attack actions).

These attacks are then combined into a single dependency structure. After this there are a number of optional steps to make this combined dependency structure more compact in the form of merging similar attacks and identifying shared sub trees.

The combined dependency structure is then converted into an attack tree with AND, OR and optionally SQAND and XOR gates. This attack tree can than be converted into any data standard required by tools.

More specific implementation details are given in the following subsections.

5.1.2 Extending GROOVE

The decision was made to implement the proposed approach as an extension of GROOVE, an existing open-source Java tool for constructing and analysing graph transformation systems. Extending GROOVE offered several advantages over developing a new standalone approach.

- GROOVE offers a GUI for specifying a GTS, and therefore allows for a natural way to construct the input models of the approach.
- Several of parts of the GROOVE implementation can be reused/extended in the implementation, such as finding all matches of rule in a graph or applying a rule transformation to the graph.
- By developing this works approach as a separate module, this offers users an additional exploration method for analysing their GTS, in addition to allowing the users the freedom to use other existing and implemented exploration options of Groove.
- GROOVE is open-source and well-maintained.
- Finally, GROOVE supports inter-operation with other tools. It supports the importing and exporting of GTSs by supporting a common data standard for exchanging a GTS.

Developing a GROOVE module The setup of Groove's implementation is very modular. The different functionalities of the tools have been divided into individual packages that each have a clear own purpose and clear dependencies on other packages.

At the core of the tool lies a package that describes what a Graph is and a package that describes GTS data-model that builds on this graph package.

Based on these foundations, there are a large number of packages for exploring a GTS. One of these is used for finding matches of a specific production rule in a specific graph and another package can be used to transform a GTS, i.e. it describes how a rule is applied to a graph and produces the resulting graph.

There are many more packages in the tools implementation, most of them describing other exploration strategies and the GUI of the tool, but these are not directly interesting for the implementation.

Our approach could base itself on and reuse most of these aforementioned packages. E.g. it can load a GTS model which was specified using the tool's GUI and then use an exploration package to find all matches.

Therefore the implementation is mostly based on correctly using the existing functionality of the tool and sometimes slightly modifying some functionality by extending it, for example by describing how a 'glue' transformation works compared to a 'apply' transformation.

5.1.3 Overview of the implementation details

5.1.3.1 Unfolding construction

The unfolding construction implementation starts with loading the input GTS, explicitly the start graph and the set of transformation rules.

Iterative unfolding construction The unfolding is then constructed in an iterative manner, with the start graph representing the unfolding for the first iteration. At the beginning of each iteration, all matches of all transformation rules in the unfolding/host graph are identified (this is performed by the existing GROOVE implementation for finding matches of a rule in a graph).

For each match of each transformation rule, each unique history is then identified. Earlier explored histories of a match, both valid and invalid, are removed from this set.

Each remaining history is then added to the unfolding. If this is the first history for the match, the match is glued to the host graph of the unfolding. After this optional step, this newly identified history is added to the elements read and created by its corresponding rule application as respectively reading and creation histories of those elements. This step makes it possible to identify new histories in the next iteration.

After adding all newly identified histories of all matches of all rules to the unfolding, the iteration is finished.

If no new histories were discovered during an iteration or if the depth restriction of the unfolding is reached, the construction is halted. Otherwise a new iteration is started with the updated unfolding as its input.

Identifying possible histories of a match The main challenge of the iteration lies in identifying all possible histories of a match. This is performed by looking at all possible histories of all elements of a match (which have been added in previous iterations).

Every possible combination of histories of elements of a match is explored. If a combination of histories is concurrent, this combination of histories represents a possible history of the corresponding match.

Determining if histories are concurrent A combination of histories is concurrent if there are no pairwise conflicts between the histories. Because these histories represent explicit sets of applied rule applications, it is possible to detect all symmetric conflicts in a pairwise manner, even if the conflict is composed out of asymmetric conflicts.

Therefore, all histories are pairwise inspected for conflicts. A conflict occurs if one history prevents another history from being applied, or if one element's history consumes the other element.

Optimisations In addition to the required functionality that guarantees that all and only all valid attacks are found, the implementation has several optimisations. These optimisations are mostly based on storing previous calculations to prevent them from having to be calculated twice. One good example of this is that each history of each element maintains sets of with what other histories of other elements this history is either concurrent or in conflict with.

5.1.3.2 Tree conversion

Once the unfolding has been constructed, the next step is to identify all the possible attacks it contains. This is done by finding all matches of the goal condition that specifies when the attacker has reached his goal. Each match of this goal condition with a valid history represents a possible attack.

These attacks are then extracted from the unfolding. After this, the general attack step version of all attack steps is identified, and the dependency structure of each attack is merged into a shared dependency structure describing all possible dependencies of all matches of the goal condition.

This shared dependency structure is then converted into an actual tree format. From this tree format, the attack tree can be exported in different formats. For now the commonly used ADTree schema[2] is used as the default output data format.

5.2 Quantitative Evaluation: Scalability comparison

5.2.1 Preliminaries

Let us start with first defining the goal of this evaluation part and the process of performing the evaluation.

The evaluation part focuses on the first step of the attack tree generation process: identifying all possible attacks from a given input model, as is shown in figure 5.1. The reason for not evaluating the second step is that it's computational complex step (reducing the size of the attack tree) is optional. As shown in Chapter 4, a basic tree representation can easily be obtained. Therefore, the

[2] http://satoss.uni.lu/members/piotr/adtool/manual.pdf

second step does not contain a mandatory bottleneck step, in contrast to the first step.

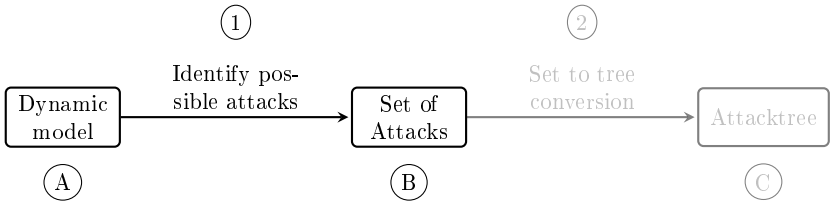


Figure 5.1: The evaluation will focus on the first step of the global attack tree generation process: identifying all possible attacks from a given dynamic model.

Specifically, the goal is to evaluate the scalability of this project's proposed approach to perform this task (constructing the unfolding of the input model) and compare this approach to the scalability of the default approach to perform this task (constructing a reachability graph) in order to determine if the approach gives rise to an improvement.

With the scalability of the approach it is meant how well the approach performs for an increase in size of the input model, e.g. how much additional computational time is required for larger input models. The scalability of the approaches is measured by applying them for differently sized input models and comparing the time required for each input model.

The comparison of the scalability of the different approaches is then performed by determining the scalability of the individual approaches using the same input models and having them perform the same task on these models. In this manner the scalability of both approaches can be fairly compared.

Because the unfolding-based approach should theoretically have an improved scalability especially for models containing a high number of concurrent actions, the scalability of the approaches is compared on two types of input models with different degrees of concurrent actions.

Let us first briefly define the steps of the two approaches that are evaluated and compared, as these steps will be measured separately in order to highlight where scalability issues arise.

5.2.1.1 Approach 1: Identifying all attacks by constructing the reachability graph of the input model

The first approach can be divided into two steps, as shown in figure 5.2. The first step is to construct the reachability graph of the input model. The second step is to extract all possible attacks from this reachability graph.

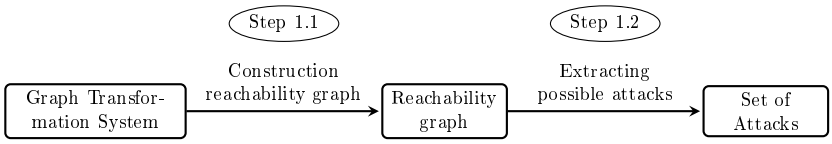


Figure 5.2: Overview of the process of the baseline approach. A reachability graph is constructed for the given model and from this graph the attacks are extracted.

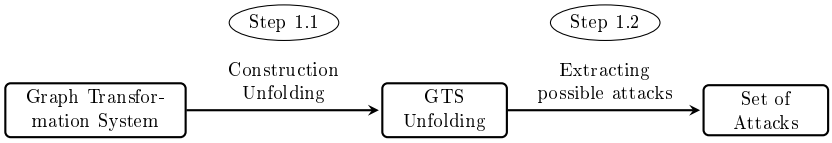


Figure 5.3: Overview of the process of the proposed approach. An unfolding of the given model is constructed and from this unfolding the attacks are extracted.

5.2.1.2 Approach 2: Identifying all attacks by constructing the unfolding of the input model

The second approach can also be divided into two steps, as shown in figure 5.3. The first step is to construct the unfolding of the input model and the second step is to extract all possible attacks from this unfolding.

5.2.1.3 Implementation of approach 1

In order to make compare both approaches in a fair manner, the existing reachability graph approach was also implemented as an extension of GROOVE so it can operate on exactly the same input models and reuse GROOVEs functionality to construct a reachability graph.

The following listing gives the implementation of this approach in pseudocode. Basically, the approach receives a GTS as input and then requests GROOVE to construct a reachability graph for this GTS. Then it performs a standard breath-first search through the resulting reachability graph to identify all sequences of events that result in a goal state. In this manner, all attacks in the model are identified.

```
1  Input = GTS describing the dynamic model.
2  Output = Set of Attacks, each consisting of basic actions
3
4  // Step 1: Construct the Reachability graph;
5  RG = GROOVE.ConstructReachabilityGraph(GTS);
6
7  // Step 2: Perform BF Search through all states
8  // and identify all sequences to goal sates.
9  BreathFirstSearch(RG, RG.getStartState().toList());
10
11 BreathFirstSearch(RG, List<State> visited){
12   Set<State> adjacent = RG.getAdjacent(visited.last());
```

```
13
14   for(State state : adjecent){
15     if (visited.contains(node)) {
16       continue;
17     } else if(state.isGoalState()){
18       visited.add(node);
19       OUT visisted;
20       visited.removeLast();
21       continue;
22     } else {
23       visited.add(node);
24       BreathFirstSearch(RG, visited)
25       visited.removeLast();
26     }
27   }
28 }
```

5.2.2 Measurements

5.2.2.1 Experimental setup

The measurements ran on a PC with an Intel i7-4800MQ CPU with 8GB of RAM. For each measurement an initial trial run was executed to allow the JIT compiler to run. The measurement itself was done on the second run. Although there is variety between runs on the same model, there was clear trend in the performance for larger models and therefore a single run measurement was deemed sufficient.

5.2.2.2 Model 1: Digital domain

The digital domain model is designed to have a low degree of concurrency between the actions. The intuition of the model is describing a digital layout of a building/organization and describing how an attacker can move between these digital locations.

The set of rules is made up by rules on how to move between digital locations based on if there is a policy or not and on how to obtain credentials to bypass policies. For this dynamic specification of the model, six different versions have been designed with differently sized static parts.

The specific version and transformation rules are omitted here for brevity, but they are all well-typed for the type graph shown in figure 5.4 and can be found in the project's public repository.

Some brief stats about the model versions:

- Version 1a: Elements: 8 nodes & 10 edges, Attacks: 10
- Version 1b: Elements: 18 nodes & 26 edges, Attacks: 56
- Version 1c: Elements: 21 nodes & 32 edges, Attacks: 68
- Version 1d: Elements: 23 nodes & 36 edges, Attacks: 92

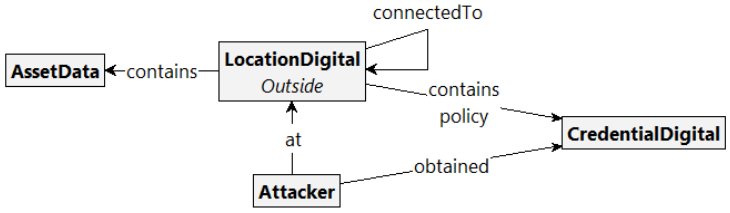


Figure 5.4: Type graph of the digital domain model

- Version 1e: Elements: 32 nodes & 52 edges, Attacks: 202
- Version 1f: Elements: 40 nodes & 69 edges, Attacks: 1084

The performance of both approaches for all versions of the digital domain input model can be seen in table 5.1. The measurements are in milliseconds and describe the time required for the approach to respectively perform the first, second and complete step for each version of the model (Model version e and f were only measured for the unfolding approach, as the reachability graph approach took to much time).

Digital domain model	Attack identification		
	Step 1.1 (ms)	Step 1.2 (ms)	Step 1 complete (ms)
1a Reachability Graph	173	4	177
1b Reachability Graph	1123	1888	3011
1c Reachability Graph	2014	17220	19324
1d Reachability Graph	3428	263666	267094
1a Unfolding	130	6	136
1b Unfolding	207	27	234
1c Unfolding	282	41	323
1d Unfolding	320	74	394
1e Unfolding	1270	163	1433
1f Unfolding	3141	205	3346

Table 5.1: Performance of both approaches on all version of the digital domain input model

5.2.2.3 Model 2: Multi-domain

The multi-domain model is designed as a model with a high degree of concurrency. It is basically an extension of the digital domain with the physical domain and adds a set of rules for relocating in the physical world (which are independent of rules for the digital world) and a single rule that connects the two domains.

Once again six versions of this model have been defined for increasing sizes, the specific rules and versions are omitted from the report and the corresponding type graph is shown in figure 5.5.

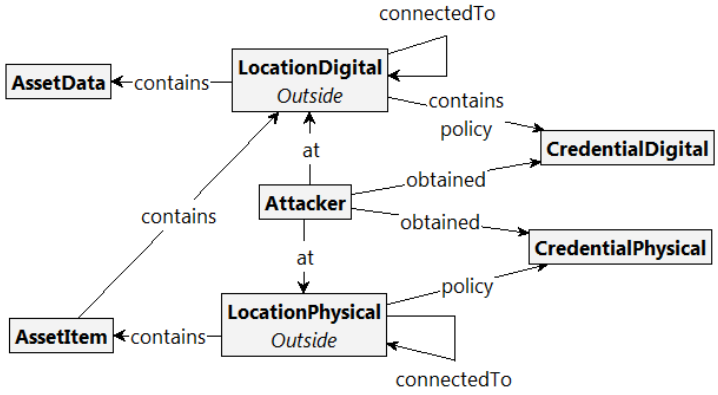


Figure 5.5: Type graph of the multi-domain model

Some brief statistics about the model versions:

- Version 2a: Elements: 9 nodes & 10 edges, Attacks: 3
- Version 2b: Elements: 12 nodes & 13 edges, Attacks: 6
- Version 2c: Elements: 15 nodes & 19 edges, Attacks: 14
- Version 2d: Elements: 23 nodes & 33 edges, Attacks: 20
- Version 2e: Elements: 30 nodes & 48 edges, Attacks: 108
- Version 2f: Elements: 43 nodes & 71 edges, Attacks: 370

The performance of both approaches for all versions of the multi-domain input model can be seen in table 5.2. Model version e and f were once again only measured for the unfolding approach, as the reachability graph approach took to much time.

5.2.3 Discussion

5.2.3.1 Model 1: Digital domain

From the performance measurement of both approaches for all four version of the digital model, as shown in table 5.1, a few things can be observed.

Both approaches perform in the same time range (100-200 ms) for the minimal version of the model. For each larger version of the model, both approaches require an increased amount of time to perform the exploration.

However, the increase in time required for each increase in model size is much larger for the reachability graph approach then for the unfolding approach,

Multi-Domain model	Attack identification		
	Step 1.1 (ms)	Step 1.2 (ms)	Step 1 complete (ms)
2a Reachability Graph	175	3	178
2b Reachability Graph	723	16	739
2c Reachability Graph	2957	7102	10059
2d Reachability Graph	3327	42937	46264
2a Unfolding	103	4	107
2b Unfolding	176	6	182
2c Unfolding	190	8	198
2d Unfolding	185	16	201
2e Unfolding	1024	87	1111
2f Unfolding	18309	120	18429

Table 5.2: Performance of both approaches on all version of the multi-domain input model

demonstrating that (as expected) the latter is more scalable then the existing approach.

The result is that for a model roughly 3 times as large (version 1 compared to version 4), the reachability graph requires 1500x more time, compared to 2.9x more time for the unfolding approach.

5.2.3.2 Model 2: Multi-domain

Similar results can be seen for the second model in table 5.2. For the minimal version of this model, both approaches start in the same performance range (100-200ms) and both require additional time as the model version grows, but the performance of the reachability graph approach decreases a lot more rapidly then the unfolding approach.

In this case, for a model version roughly twice the size (but with a lot more concurrent actions), the reachability graph requires 260x more time, while the unfolding approach requires roughly 1.9 times more time to explore the fourth version of the model.

For the larger version of the model, versions e and f, there is however a significant increase in the time required to identify all possible attacks.

5.2.3.3 Models compared

A comparison of the performances of the approaches for the different models is more difficult, as the models have different properties, such as the amount of concurrency and amount of attacks they contain.

It can however be seen that the performance of the reachability graph approach follows a similar trend for both models. The time required to identify all

possible attacks increases rapidly for larger input models, in both cases much more then this project's approach.

We noted before however that a model with a large degree of concurrency is expected to benefit more from partial-order reduction, but during the evaluation of the approach for the largest two versions of both models, the opposite effect appears to be seen. I.e. the approach for the Digital Domain appears to perform better then for the multi-domain, even though that model contains more concurrent actions.

Closer inspection reveals the reason for this. For the multi-domain model, there are on average much more histories for each rule application that need to be explored. This causes a drop in performance, as the current implementation makes exploring histories expensive operations.

Therefore, it appears that having a lot of concurrent actions results in a lot of possible histories that has an impact on the performance (but not nearly as much as without using partial-order reduction).

The digital domain seems to be a good fit in between the two extremes. By having a medium amount of concurrent actions, it can enjoy the benefit of the partial-order reduction, but is not burdened by a large amount of possible histories.

We believe however that the performance of the implementation for exploring possible histories can be significantly improved, as was also mentioned in Chapter 3, so that the approach performs better for models with a large degree of concurrent actions.

5.3 Qualitative Evaluation: Cloud Case Study

The goal of this section is to evaluate several individual aspects of the approach and the entire process itself.

Specifically, this section evaluates:

1. How a practical, existing, security-domain specific problem can be modeled using graph transformation.

2. How the output of the attack tree generation procedure, an attack tree, can then be analysed.

3. Evaluate how generic the approach is by introducing a change scenario to the input modeling language.

This evaluation part is performed by applying the implemented approach to an existing case study and observe the interesting parts.

5.3.1 The Cloud Case Study

The cloud case study focuses on a data center facility that contains a file of which the content is of a sensitive nature. This file is located within a virtual

machine which runs on a (physical) server that is located in a server-room of the data-center, and there is an attacker interested in accessing this file.

The version of the case study used in this chapter is a simplified variant of a case study obtained from the TREsPASS research project. The case study provides a description of an organization in the form of a model and requests the generation of attack trees for this organization.

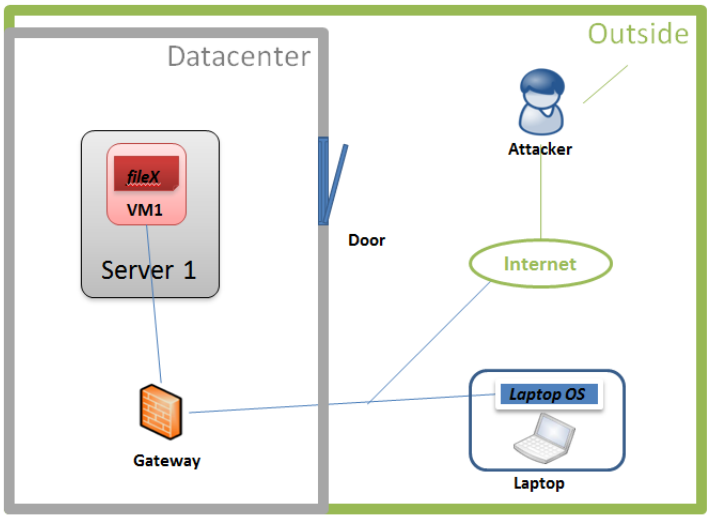


Figure 5.6: Simplified TREsPASS model of the Cloud Case Study

An informal visualisation of the cloud case can be seen in figure 5.6.

Because the TREsPASS research project focuses on security covering multiple domains, a domain specific modeling language for modeling the security aspects of the physical, digital and social security domains has been developed within this project. Therefore, the case study description/model has also been developed in this so-called 'socio-technical modeling language'.

Even though the attack tree generation approach proposed in this work is independent of specific modeling concepts (and therefore of the modeling language itself), it is beneficial to understand how the security aspects of a system or organization might be modeled and analyzed using graph transformation. Therefore, the modeling language developed in the TREsPASS project and used in the case study is used as an example of how a system or organization might be modeled in practice. This section demonstrates how this model can also be modeled using graph transformations and how this model can then be analyzed to extract all attacks and convert them to an attack tree.

For this reason, the socio-technical modeling language itself is also introduced in this section.

5.3.1.1 Socio-technical modeling language

The modeling language is based on a number of domain specific concepts that are used to describe the security properties of a system or organization in an abstracted manner.

The language contains concepts for three different domains in the form of Physical, Digital and Social security. As the language is still in development, the concepts introduced are subject to change.

Static description: Organization layout At the core of the modeling language lie five concepts:

1. Location/Physical
2. Location/Digital
3. Asset/Item
4. Asset/Data
5. Attacker (unique)

In addition to these concepts, there is a set of rules specified on these concepts to complete the core of the proposed model.

- An Asset/Data is always contained by a Location/Digital.
- An Asset/Item is always contained by a Location/Physical.
- An Asset/Item can contain a Location/Digital (e.g. a server containing a VM).
- Locations of the same type can be connected to each other.
- The Attacker is assigned both a Physical as well as a Digital starting location.

Example - The earlier shown visualisation of the cloud case (figure 5.6) can also be described using this set of concepts. Table 5.3 classifies all entities en relations of the previous visualisation as one of these concepts.

To prevent any confusion, we see a 'Server' as simply being a physical object. The virtual machine VM1, which is the piece of software that runs on this server, is modeled separately as a digital location that is contained by this physical object. While in the real world these concepts are more intertwined (the server may run its own OS for example) we use abstract versions of them to obtain a simplified model. This is also why Laptop and Laptop OS are modeled as separate concepts, as each have different properties for the generation part.

Concept	**Name**	Contains	*Connects*
Asset/Data	**FileX**		
Location/Digital	**VM1**	FileX	*Gateway*
Asset/Item	**Server1**	VM1	
Location/Digital	**Gateway**		*VM1, Laptop, Internet*
Location/Physical	**Datacenter**	Server1, Gateway	*Door*
Location/Digital	**Laptop OS**		
Asset/Item	**Laptop OS**	Laptop OS	*Gateway, Internet*
Location/Physical	**Door**		*Datacenter, Outside*
Location/Digital	**Internet**		*Gateway, Laptop OS*
Location/Physical	**Outside**	Door, Internet, Laptop	*Door*
Actor	**Attacker**		*Physical: Outside, Digital: Internet*

Table 5.3: Concepts of the simplified TREsPASS model

Static description: Organizational policies The basic concepts are used to describe the layout of an organisation and show how these objects relate to each other to form the basis of a model.

In addition to the layout of a model, the policies used within the organization that prevent assets from being stolen/accessed are also relevant and therefore included in the model. Examples of policies range from a door that requires a key to a gateway that only allows connections with a limited set of IP addresses.

Policies are specified using the following concepts

1. Policy
2. Credential/Digital
3. Credential/Physical
4. Role

All locations in the model can have a policy that specifies a requirement to access that location. A policy specifies its requirement in the form of a credential. A credential can be either a digital credential such as a password, or a physical credential such as a key.

Finally, the model specifies where these credentials may be obtained. This is done introducing the concept of roles in an organization that may have different credentials (e.g. cleaning crew has different credentials than the owner) in addition to locations that may contain credentials (such as a room containing a key).

Based on these new concepts, we can then specify what roles an organization has, what credentials each role has and finally what credentials are required for each policy within the model.

In table 5.4 the case study example is extended with these additional concepts to demonstrate how they can be combined.

Table 5.4: (extended example) Concepts of the simplified TREsPASS model

Concept	**Name**	Has credential	*Policy*
Asset/Data	**FileX**		
Location/Digital	**VM1**	Valid IP	*Requires password VM*
Asset/Item	**Server1**		
Location/Digital	**Gateway**		*Requires Valid IP*
Location/Physical	**Datacenter**		
Location/Digital	**Laptop OS**	Valid IP	
Asset/Item	**Laptop OS**		*Requires password Laptop OS*
Location/Physical	**Door**		*Requires Key*
Location/Digital	**Internet**		
Location/Physical	**Outside**		
Actor	**Attacker**		
Credential/Physical	**Key**		
Credential/Digital	**Password VM**		
Credential/Digital	**Password Laptop OS**		
Credential/Digital	**Valid IP**		
Role	**Cleaner**	Key	
Role	**Maintenance**	Key, Password VM	
Role	**Boss**	Key, Password Laptop OS, Password VM	

Dynamic description: Attacker actions In addition to a set of concepts to define the static description of an organization, the Socio-Technical modeling language also has a set predefined attacker actions.

These actions are purposefully very generic. Domain-specific attacks are left out for later refinement. Therefore the attacker has capabilities such as 'Obtain Credential from Role' or 'Use Credential to access Location'. In the TREsPASS project, a separate attack pattern library (APL) is developed that contains so-called attack patterns with domain-specific attacks. After generating the attack

tree, the APL is used to refine the generated tree with domain-specific attacks, such as the possibilities to either bribe, threaten or steal to obtain a certain credential from a role.

Below, the set of possible attacker actions is specified in an informal manner.

The attacker can:

- Move to a Location/Physical that is connected to the Location/Physical the attacker currently resides. If the second Location/Physical requires a policy, the required credentials must be supplied or the policy must be bypassed somehow.

- Gain access to a Location/Digital that is connected to the Location/Digital it currently has access to. If the second Location/Digital requires a policy, the required credentials must be supplied or the policy must be bypassed somehow.

- Access an Asset/Item if it is contained by the Location/Physical the attacker currently resides.

- Gain access to a Location/Digital if it has access to the Asset/Item that contains this Location/Digital.

- Access an Asset/Data if it is contained by a Location/Digital the attacker currently has access to.

- Obtain a Credential from a Location the attacker has access to (Digital) or currently resides in (Physical).

- Obtain a Credential from a Role.

5.3.2 Modeling the cloud case as a GTS

This section will demonstrate how the cloud case can be modeled as a graph transformation system that can be used as input for this project's attack tree generation approach.

In order to define the cloud case as a model, first the socio-techinical modeling language is modeled in a GTS.

5.3.2.1 Defining the static part: Layout and Policies

The static part of the socio-technical modeling langauge can be defined as a type graph of a graph transformation system, as given in figure 5.7.

At the core of this type graph lays the Attacker. The attacker has access to a physical location and a (set of) digital locations. The layout of the organization can be described by digital and physical locations that are connected to each other, item assets contained by physical locations, data assets contained by digital locations and digital locations contained by item assets.

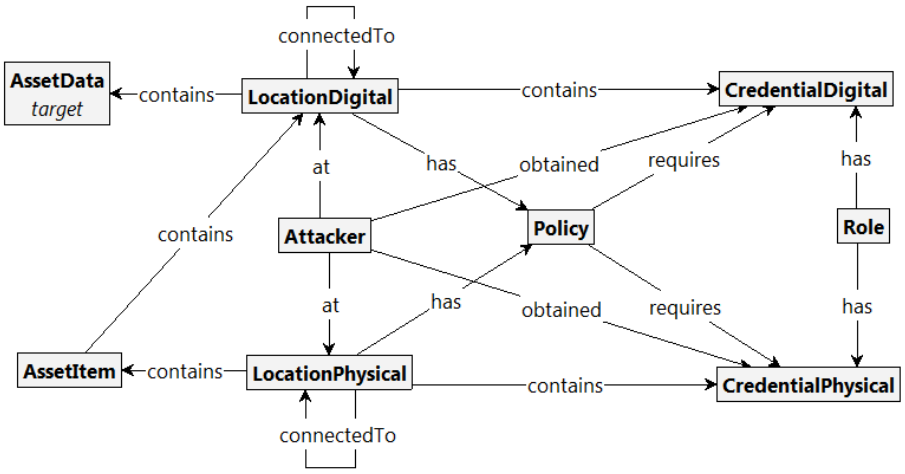


Figure 5.7: Type graph representing the entities of the socio-technical modeling language

Both types of locations can, however, also have a policy. A policy has a set of required credentials. These credentials are connected to the different roles of an organization, but the attacker can obtain each credential.

Finally, locations can also contain credentials.

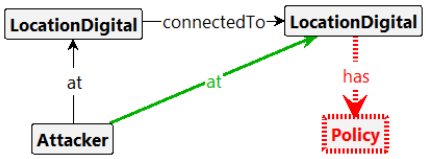
5.3.2.2 Defining the dynamic part of the model: Attacker actions

The set of attacker actions of the model, or the dynamic part of the model, is defined by a set of eleven transformation rules shown in figure 5.8. The first three rules specify the different ways the attacker can gain access to new digital locations; namely simply move if the targeted location does not have a policy, or either bypass the policy or use the required credentials if the targeted location has a policy. The fourth and fifth rule, figures 5.8d and 5.8d, specify how the attacker can obtain credentials, either from any defined role that has these credentials or from a location that contains them.

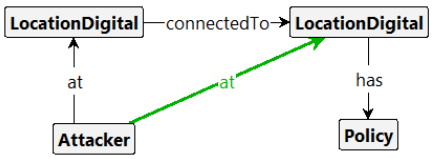
The following five transformation rules (figure 5.8f, 5.8g, 5.8h 5.8i and 5.8j) specify the same actions but then for the physical world. There is one big difference to observe and that is that in the physical world, the attacker can only be at one location at the same time. Therefore the physical rules all remove the 'at' arrows, in contrast to the digital rules.

Finally, the last transformation rule (figure 5.8k) specifies how the attacker can use his access to a physical location to gain access to a digital location.

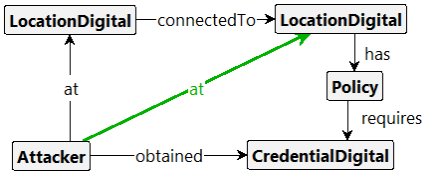
One interesting thing to observe is that both the 1st and 6th rule (figure 5.8a and 5.8f) specify NACs, even though this work does not fully support NACs. This does not mean that all usages of NACs are prohibited. As long as the element described in the NAC, in this case the existence of the 'has' edge to



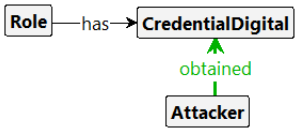
(a) Digital move



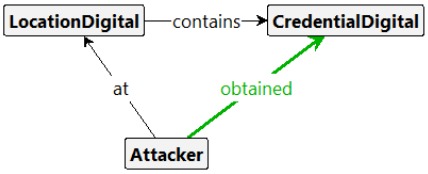
(b) Digital bypass credential



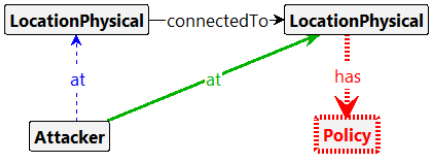
(c) Digital use credential



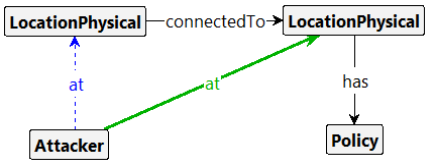
(d) Digital obtain credential



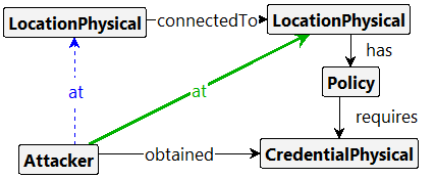
(e) Digital obtain credential contained by location



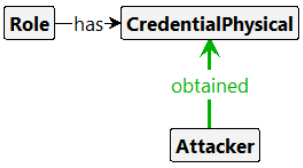
(f) Physical move



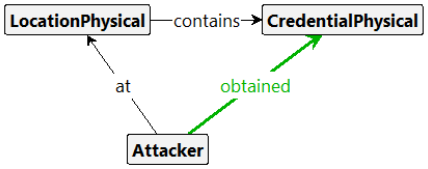
(g) Physical bypass policy



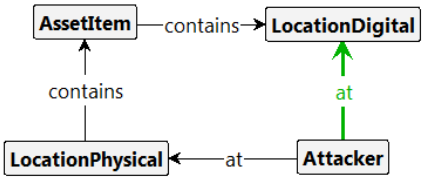
(h) Physical use credential



(i) Physical obtain credential



(j) Physcial obtain credential contained by location



(k) Reach Digital location through Physical access

Figure 5.8: The transformation rules representing the attacker actions

a policy, cannot be added or removed in the model, there is no effect on the attack identification procedure by this type of NAC usage.

5.3.2.3 Defining the specific example

Finally, the specific cloud case example can then be modeled as an instance of the type graph, representing the graph of a specific GTS.

Figure A.1 (see the Appendix) shows the graph modelling the specific cloud case example defined earlier. This graph, in combination with the set of transformation rules, forms the GTS that is then given as input to the attack tree generation approach.

5.3.3 Analysing the resulting tree

The goal of this section is to demonstrate how the trees generated by our approach can then be used for analysis by existing tools.

5.3.3.1 Generating the attack tree

The attack tree generation approach can be given the cloud case GTS, defined in the previous subsection, as its input model and will construct the corresponding attack tree. It can identify 40 different possible attacks for this given input model.

Due to size limitations of this report, the entire tree is not specified. However, to give an indication of how the resulting tree looks like, an attack tree for a subset of nine of the 40 possible attacks is given in figure A.2 in the appendix.

As mentioned previously, the constructed tree can be exported to different data formats, with the default being ADTree schema. ADTree schema is the data format of ADTool, a tool for visualising and analysing attack trees. The visualisation of the attack tree in the appendix is generated using ADTool.

5.3.3.2 Qualitative and Quantitative analysis

Qualitative analysis can be done by loading the model directly into ADTool and manually analysing the tree. ADTool has features to zoom-in on interesting parts and hide other parts of the tree to assist in this process.

Assigning values to basic actions. The quantitative analysis requires that all basic actions/leafs are annotated with values. There are multiple manners in which this can be done.

- Use ADTool to assisgn values to all leafs by hand. ADTool can then also be used to perform simple calcualtions on the tree.
- The TREsPASS project an APL (Attack Pattern Library) can be used for refining/annotating generated trees with domain-specific info, including values for basic actions.

- A third option is to assign values to actions in the GTS, so that the generated tree already contains the values. This may also have some additional benefits, such as being able to halt the exploration if the probability of a certain attack happening becomes too small.

Quantitative analysis The goal of this section is to show how the generated attack tree can then be used by a quantitative analysis tool. For this example the ATCalc tool [42] is chosen as this tool supports attack trees with SQAND gates. The values for the basic actions (in this case probability of success and its relation to time) were manually added to the attack tree in ADTool.

After annotating the attack tree in ADTool, it was exported and converted to ATCalc input using the Attack Tree Transformation (ATT)[3] project which facilitates the transformation of attack trees to different formats using model transformations.

The resulting ATCalc input is shown in the appendix, see listing A.1. From this input it can be seen how the difference between sequential ANDs and regular ANDs is encoded.

In addition to the attack tree, ATCalc requires users to specify the question they are interested in. For this example, the increase in probability of the attacker reaching his goal.

The online ATCalc interface[4] then generates a plot that shows this increase in probability over time, which is shown in figure 5.9.

The specific values that are annotated to the tree and the specific values of resulting plot is not the point of interest. The main point is to demonstrate that the attack trees generated by this project's approach can directly be annotated and analysed by existing tools.

5.3.4 Evaluating genericity by introducing a change scenario

Now let us image that an organization decides to implement a countermeasure in the form of an alarm in their organisation. When the company detects a bypass of a policy (and for now let us assume that they always detect this action), the alarm will be triggered and all policies will no longer accept any credentials.

The organization now wants an updated attack tree that takes into account this countermeasure in determining what the possible attacks are. Therefore, this countermeasure should be put into the model of the organization. The existing socio-technical modeling language, however, does not support the specification of such a countermeasure and therefore needs to be extended.

As a sidenote: If an attack tree generation approach commits itself to the socio-technical modeling language and optimizes its exploration for this input model, the exploration will also have to be redefined for this alteration of the modeling language. This countermeasure is also a good example of an event that

[3] https://github.com/djhuistra/UnifyingAttackTrees
[4] http://fmt.ewi.utwente.nl/puptol/atcalc/

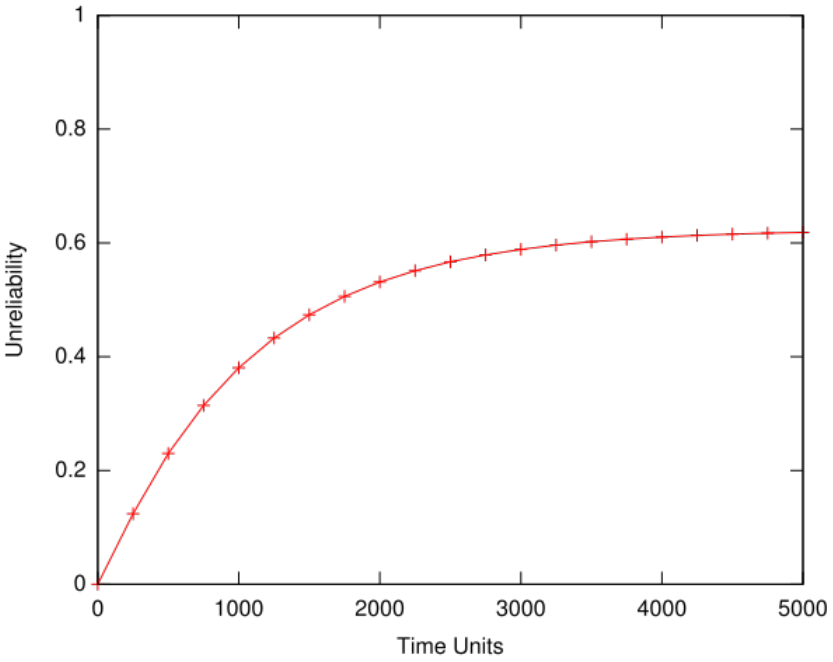


Figure 5.9: ATCalc output graph showing the probability over time of the attacker reaching his goal. Unreliability (on the y-axis) in this case means Probability of succes.

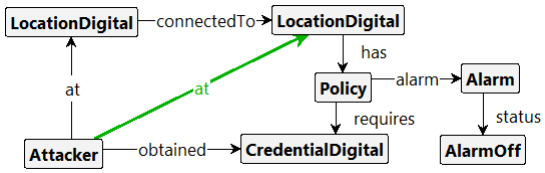
does not fall under the monotonicity assumption. Previous events triggering the alarm will affect future events.

To determine if this project's proposed approach is generic enough to support this kind of extension, the earlier defined GTS for the cloud case study is updated with this countermeasure.

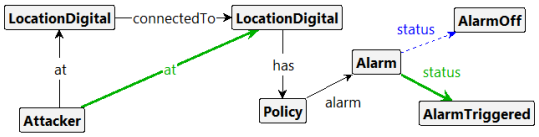
Updated GTS To include this change scenario, the input model is updated both in the static part and the dynamic part. The static part describes the alarm and the policies it is connected to, while the dynamic part describes when the alarm is triggered and what its effect is on future attacker actions. The updated and additional transformation rules after implementing this change scenario are shown in figure 5.10.

The updated digital *use credential* rule checks if the alarm is not yet triggered. Only when this is the case will it accept the credentials. The *updated bypass* policy rule describes that when a policy is bypassed, the alarm should be triggered. An additional *digital bypass policy* rule describes that once the alarm is triggered, the attacker can still attempt to bypass the policy to gain access to additional digital locations.

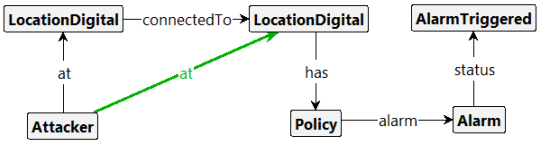
From this set of transformation rules, the type graph and graph of the updated cloud case model can be inferred and are therefore omitted here.



(a) Updated digital use credential



(b) Updated digital bypass policy



(c) Second digital bypass policy

Figure 5.10: The updated set of transformation rules for the alarm change scenario

Attack tree generation for the Updated GTS The updated GTS can then be used as a new input model (using different concepts) of the attack tree generation approach. By specifying the change scenario using standard graph transformation concepts, the approach for identifying all possible attacks does not need to be altered to support this updated GTS.

There is however a difference in the resulting attack tree in the form of a reduction in the number of possible attacks found in the exploration because of this change scenario. This, of course, is the expected result of adding the countermeasure

5.3.5 Discussion

In this section, the complete approach was evaluated by applying it to an existing case study.

The first task was to model the case study (and the socio-technical modeling language it is specified in) as a graph transformation system. The language was modeled by developing a type graph to specify the static part of the models and a set of transformation rules to specify the dynamic part. A specific start graph then specifies the cloud case example, a textual program in the language.

With this step, it was shown that the graph transformation modeling paradigm is expressive enough to specify socio-technical models. The main observation was that supporting negative application condition would increase the expressiveness.

The second step was to apply the approach to this input model and study the result. It was manually checked if the resulting attack described all expected attacks and no others in order to confirm the *exhaustive* and *succinct* properties.

In addition, it was also demonstrated how the resulting attack tree can then be used for analysis. There are multiple options for qualitative analysis, just as there are multiple tools for this. The decision was made to manually annotate the tree in ADTool and then transform the tree into ATCalc input and asks this tool to calculate the expected probability of success of the entire tree over time.

Using (the concepts of) the ATT project, it is expected that the resulting attack tree can be transformed into the input format of most analysis methods. The main challenge in this part lies with annotating the trees with sufficient information for each analysis to operate on.

The third part was to demonstrate that the approach is generic, by introducing a change scenario to the cloud case that cannot be expressed in the socio-technical modeling language. It was demonstrated that our approach is capable of supporting this change without alteration. This property should be maintained in general. As long as something can be specified in a GTS, it can be used as the input for this works proposed approach.

Limitations of the evaluation There are however several limitations to the result of this evaluation. The main limitation is that the author was the only tester. We believe it would be beneficial to have the approach evaluated by

domain experts, for example to determine if graph transformations are truly a suitable modeling approach and if the generated attack trees are truly suitable for analysis. This step was however omitted due to time limitations.

Another limitation is that the approach has not been evaluated in comparison to TreeMaker, another tool developed within the TREsPASS project to generate attack trees that commits to the socio-technical modeling language. The reason for this is that we could not decide on a fair comparison between the two approaches. While both approaches should identify the same set of attacks in the input model, TreeMaker also uses its knowledge about the concepts in the input language to construct the attack tree. The resulting attack trees are therefore very different from the trees our generic approach is capable of producing. This makes the results difficult to compare.

Chapter 6

Related Work

The purpose of this chapter is to give an overview of the related work performed on the subject of analyzing models to identify all potential attacks. This issue has not only been investigated for attack trees as the target model: There are a number of efforts that produce the results in different formats, such as an attack graphs or attack routes. In addition in this chapter we will comment on the differences between these efforts and the approach described in this thesis.

The initial research on analyzing a dynamic model to identify potential attacks was performed in 2002 by Sheyner *et al.* [9], who focused on the automated generation of an attack graph, similar to a reachability graph, for the network security domain. They demonstrated how existing symbolic model checking algorithms can be used to generate this attack graph, but also commented on the state-space explosion that occurred for larger input models.

Several efforts within the network security domain have attempted to improve upon this work. Both Ammann *et al.* [14] and Ou *et al.* [10] have proposed a polynomial bounded algorithm to construct the attack graph, but they rely on the assumption of monotonicity of attacker actions and on the absence of negation to obtain this drop in complexity.

Hong *et al.* [43] have proposed a method that uses logic reduction techniques to directly construct an attack tree based on a given network system, but this effort basically rewrites the representation of the input into an attack tree instead of identifying possible attacks, and therefore does not perform any dynamic exploration using attacker actions.

More recently there have been a number of approaches for domains other than network security.

Dimkov *et al.* [16] have developed an analysis method to identify attack scenarios for their security framework that combines three security domains (physical security, digital security and security awareness). Their analysis method uses the work by Ammann *et al.* in order to analyze the model with a polynomial bounded algorithm and then improves upon this by removing the overestimated attacks using modeling language specific concepts to identify these attacks.

Ivanova *et al.* [12] have developed a custom analysis method for their so-called *socio-technical modeling language* that uses its knowledge about the concepts specified in the input language to perform directed searches, but they have not commented on the performance of their method.

Vigo *et al.* [13] have proposed a static analysis approach based on the idea of using a process algebraic specification as the input model that is translated into sets of logical formulae to automatically infer an attack tree. While they avoid resorting to a specification language tailored to a specific domain, so that the approach can be used to model a great many scenarios, the authors have confirmed in private communication that their approach can lead to an overestimation of the potential attacks, as their idea of discovering secret channels as a way to represent obtaining credentials is similar to the monotonicity assumption as these channels cannot be lost.

Pinchinat *et al.* [11] have proposed an approach in which an expert can participate in the tree construction process by specifying so-called high-level actions that are used to merge similar attacks. Their approach is based on constructing an attack graph through using model-checking techniques. The input of their approach is a (visual) building specification which is generated into a modeling formalism. Their work is implemented as a tool built on top of Eclipse [44]. While they do not comment on the performance of their approach, they acknowledge the scalability problems in generating attacks but mention that users can directly tune the input to scope the generation.

Chapter 7

Conclusion

> It's more fun to arrive a conclusion than to justify it.
>
> Malcolm Forbes

This final chapter first discusses this work's contributions. This is followed by the general conclusion of this work and an overview of the ideas for future work.

7.1 Discussion

The goal of this discussion section is to discuss the proposed attack tree generation approach in its entirety and specific parts of the approach individually to highlights its strengths and weaknesses.

Part 1: Identifying all possible attack from an input model This work's main contribution was to demonstrate that partial-order reduction can be used to improve the scalability of the attack tree generation approach described in previous efforts without limiting the genericity of the approach to specific (security-)domains.

An existing technique for partial-order reduction that unfolds a graph transformation system was used to implement this step of the process and evaluated to show that this results in an improved scalability.

The current implementation does, however, have some limitations. Mainly it does not support the creation or removal of nodes and negative applications conditions by the transformation rules. In addition to this is the fact that the approach does not detect cycles behavior in the input models and when input models contain such cycles, this reduces the performance of the approach.

We believe however that all of these limitations can removed and gave pointers as to how this might be achieved. In the meantime however, the unfolding approach simply offers an alternative exploration method that provides a potential increase in scalability (depending on the degree of concurrent actions)

for a subset of the possible input models. For models not in this subset, the reachability graph approach based constructing a reachability graph can still be used to identify all possible attacks (with the exception of infinite behavior models).

A final word about the improved scalability of the unfolding approach. While we have demonstrated that constructing the unfolding has (the potential to) significantly increase the scalability compared to constructing a reachability graph, there are still many optimizations left by which the performance may be increased. The authors of [19] mention a number of possible optimizations to improve the efficiency of the filtering of invalid matches as they acknowledge that this step will be the main bottleneck of the unfolding construction approach.

Part 2: Convert a set of attacks into a tree representation This work has also shown how a set of attacks can be converted into a tree representation. The main contribution of this step was to show how sequential information, obtained from the unfolding of the input model, can be put into the tree to provide additional analysis opportunities. In principle it is very easy to generate a basic tree representation of the attacks that simply lists all attacks as separate options to obtain the goal.

In addition to this, this work has also shown how to reduce the size of the tree. This step reduces the computational resources required to analyze an attack tree. While the proposed method to reduce the size of the tree does not guarantee any optimal representation or even a small representation, its main contribution is to demonstrate how the sequential information can be maintained during the optimization of the tree in a low-cost computational manner in order not to burden the performance of the entire process.

This tree optimization method can however be replaced by a number of different existing options to optimize the attack tree representation. It often depends on the goal of the user what the best option is in this case, e.g. depending what analysis tool he/she wants to use to analyze the tree.

Cloud Case study During the evaluation of the approach by applying it to the cloud case study, it was found that this case study, which focuses on combining multiple security domains into a single input model, is a good showcase for this work's proposed approach, as the input models have a high degree of concurrent actions (because actions are for different security domains). Therefore, the proposed approach can make optimal use of this concurrency to reduce the exploration required and improve the performance of the approach.

In addition, it was demonstrated that the graph transformation modeling paradigm has sufficient expressiveness to model an existing, security-domain specific modeling language.

Finally, it was demonstrated that the approach is generic by introducing a change scenario to the input models and showing that the approach did not need to be altered to support this change.

7.2 Conclusion

The main contribution of this work has been to demonstrate that partial- order reduction can be used to improve the scalability of the task of identifying all possible attacks from a given input model describing an organization. The scalability improvement is shown in comparison to the existing approach to perform this task based on constructing a reachability graph/state-space of all model states.

In contrast to other approaches to improve the scalability of this task, namely optimizing the model analysis for a domain-specific modeling language or reducing the exploration required by assuming monotonicity on the attacker steps, this work's approach, viz. building the unfolding of a graph transformation system, is (security-)domain independent and can therefore be reused for different domains, even outside of the security domains.

In addition, this work has demonstrated that the graph transformation modeling paradigm has sufficient expressiveness to describe (the security aspects) of an organization. This has been evaluated by using it to express an existing domain-specific modeling language and an existing case study and it is therefore deemed suitable as a generic input modeling language of a generic attack tree generation approach.

Using graph transformations as a modeling paradigm also allowed the large body of research on this subject and existing (partial) implementations to be reused. In particular the unfolding constructing for graph transformation systems was used to implement the partial-order reduction exploration of the input model and GROOVE was used as an implementation supporting traditional graph transformation features, such as defining a model though a GUI and identifying possible matches of transformation rules in a graph.

While this work's implemented approach currently only supports a limited set of the graph transformations modeling paradigm's features, with the main missing feature being support for NACs, it is expected that support for these features can be added in the future.

In addition, the performance of the approach is significantly reduced when the input model contains cycles. This can be improved by implementing cycle detection in the approach.

The third contribution of this work was to demonstrate that partial-order reduction approach can be used to provide additional information to the attack tree conversion approach to include sequential and exclusive relations about combinations of attack steps.

Finally, this work describes and implemented a complete approach for generating attack trees that can be used by analysis tools. Only the quantitative evaluation of the trees requires an additional, often domain-specific step of annotating all basic actions with values, but suggestions for this step have also been given, including a way to specify this information into the input model.

7.3 Future work

This section gives an overview of the main opportunities to continue this work. It is divided into two categories: Extending the functionality of the approach and security-domain improvements.

Extending the generic functionality The main contribution to the functionality of the approach would be to extend the unfolding approach for all possible input models, i.e. adding support for models that remove and create nodes, models using negative application conditions and models who have cyclic behavior. Adding support for models that remove and create nodes is purely an implementation issue. Support for the other types of models can be based on research on the topic of Petri nets.

In addition to adding support for these type of input models, the performance of the approach can still be optimized in a number of ways, mainly on the filtering of invalid matches found during the unfolding which is considered to be the main bottleneck of the approach according to Baldan *et al.* [19].

Finally, the approach could be extended with different options for optimizing the attack trees. For some purposes, such as a subset of the possible analysis tools, sequential and gates and exclusive or gates are not used or supported, and therefore alternative optimization options should be used to reduce the size of the tree. One option would be to convert a tree into a disjunctive normal form, for example.

Adding security-domain functionality In addition to extending the generic functionality, there are also opportunities to make the approach more useful and user-friendly for the security domain.

One clear example of this is that the approach currently generates hard-to-read basic action descriptions. This could be improved by provide domain-specific hooks; e.g. by allowing users to specify what the basic action labels should be. This could be implemented by allowing users to specify a template for each transformation rule, and then use the Id's of nodes of the match of the transformation rule to generate each instance of the template. For example, a 'Digital move' transformation rule could specify 'Digital move from $par0.id$ to $par1.id$' as its template, an instance of this for a specific match of the transformation rule could then be 'Digital move from Inside to Datacenter', which is a lot more readable then the currently generated 'Digital move[n0;n2;n5]' for example.

Another example is to provide functionality to automatically generate the value annotations of basic actions, such probability of success or average time to completion, so that the generated trees can directly be analyzed by analysis tools. One option for implementing this would be to include the required information in the input model, e.g. assigning values to rule transformations.

Finally, it should be studied further (through user studies) if graph transformations are a good modeling language to specify organization in practice. I.e. if

this modeling paradigm is both intuitive and user friendly as well as expressive enough to specify all required concepts.

Appendices

Appendix A

Cloud Case Study

Listing A.1: Attack tree shown in figure A.2 converted to ATCalc input

```
1
2 toplevel "goal[n0;n2;n3]";
3
4 "goal[n0;n2;n3]" or  "SQAND-0" ;
5 "SQAND-0" seqand  "XOR-1"  "DMBP[n0;n5;n2;n9]" ;
6 "XOR-1" or  "DMBP[n0;n1;n5;n6]"  "SQAND-1"  "SQAND-2"  "SQAND-4" ;
7 "SQAND-1" seqand  "OR"  "DMBP[n0;n4;n5;n6]" ;
8 "OR" or  "DM[n0;n1;n4]"  "PDR[n0;n12;n18;n4]" ;
9 "SQAND-2" seqand  "XOR-2"  "DMWC[n0;n7;n1;n5;n6]" ;
10 "XOR-2" or  "DOCR[n8;n7;n0]"  "SQAND-5" ;
11 "SQAND-5" seqand  "OR"  "DOCC[n0;n4;n7]" ;
12 "SQAND-4" seqand  "XOR-3"  "DMWC[n0;n7;n4;n5;n6]" ;
13 "XOR-3" or  "AND-1"  "AND-2" ;
14 "AND-1" and  "DOCR[n8;n7;n0]"  "DM[n0;n1;n4]" ;
15 "AND-2" and  "SQAND-5"  "DM[n0;n1;n4]" ;
16
17 "DMBP[n0;n1;n5;n6]" lambda=0.002 dorm=0 prob=0.15;
18 "DM[n0;n1;n4]" lambda=1.0 dorm=0 prob=1;
19 "PDR[n0;n12;n18;n4]" lambda=0.01 dorm=0 prob=0.5;
20 "DMBP[n0;n4;n5;n6]" lambda=0.0025 dorm=0 prob=0.2;
21 "DOCR[n8;n7;n0]" lambda=0.001 dorm=0 prob=0.4;
22 "DM[n0;n1;n4]" lambda=1.0 dorm=0 prob=1;
23 "PDR[n0;n12;n18;n4]" lambda=0.01 dorm=0 prob=0.5;
24 "DOCC[n0;n4;n7]" lambda=0.1 dorm=0 prob=0.9;
25 "DMWC[n0;n7;n1;n5;n6]" lambda=0.1 dorm=0 prob=1;
26 "DOCR[n8;n7;n0]" lambda=0.001 dorm=0 prob=0.4;
27 "DM[n0;n1;n4]" lambda=1.0 dorm=0 prob=1;
28 "DM[n0;n1;n4]" lambda=1.0 dorm=0 prob=1;
29 "PDR[n0;n12;n18;n4]" lambda=0.01 dorm=0 prob=0.5;
30 "DOCC[n0;n4;n7]" lambda=0.1 dorm=0 prob=0.9;
31 "DM[n0;n1;n4]" lambda=1.0 dorm=0 prob=1;
32 "DMWC[n0;n7;n4;n5;n6]" lambda=0.1 dorm=0 prob=1;
33 "DMBP[n0;n5;n2;n9]" lambda=0.001 dorm=0 prob=0.35;
```

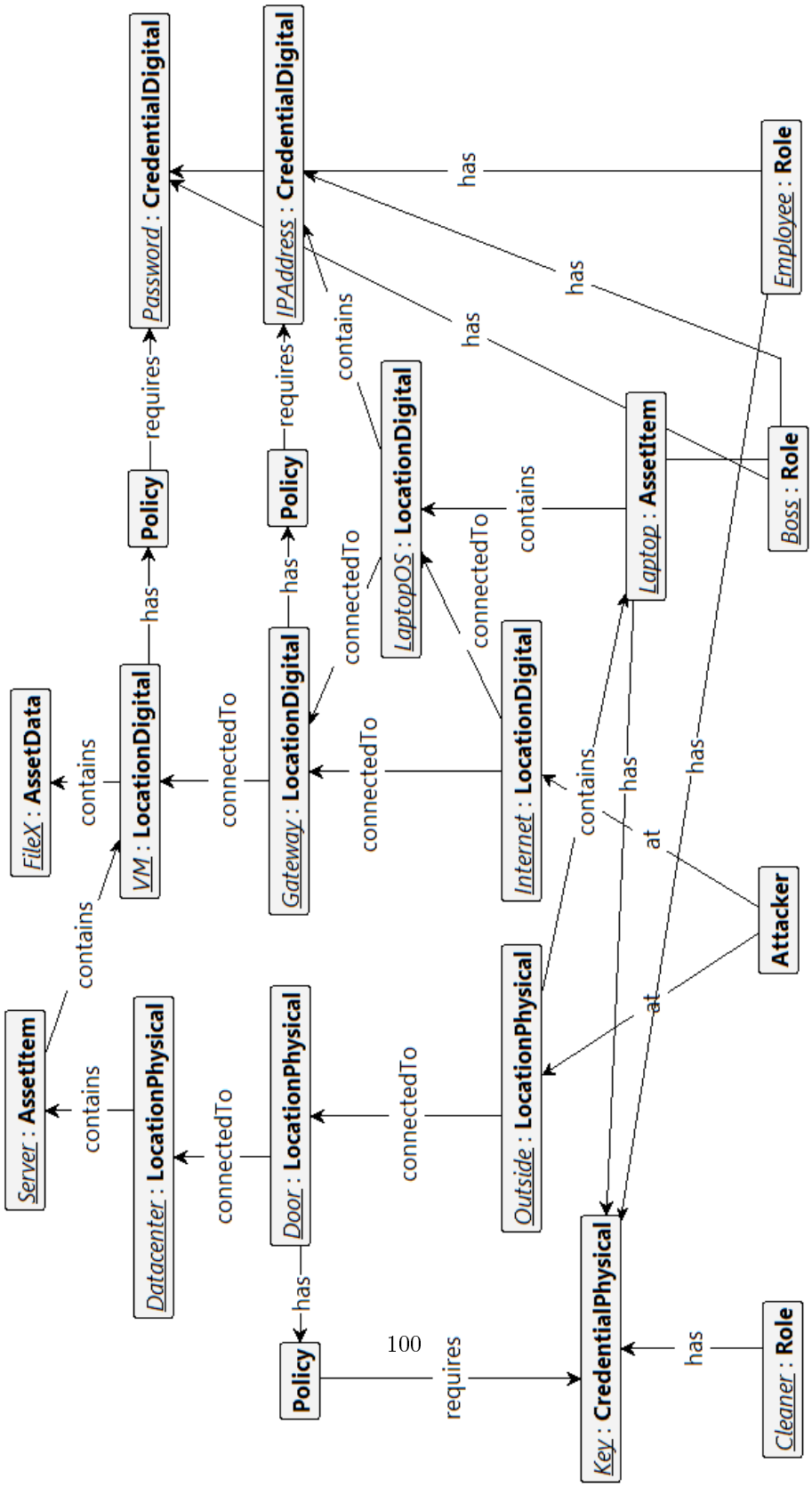


Figure A.1: Start graph of the specific cloud case example

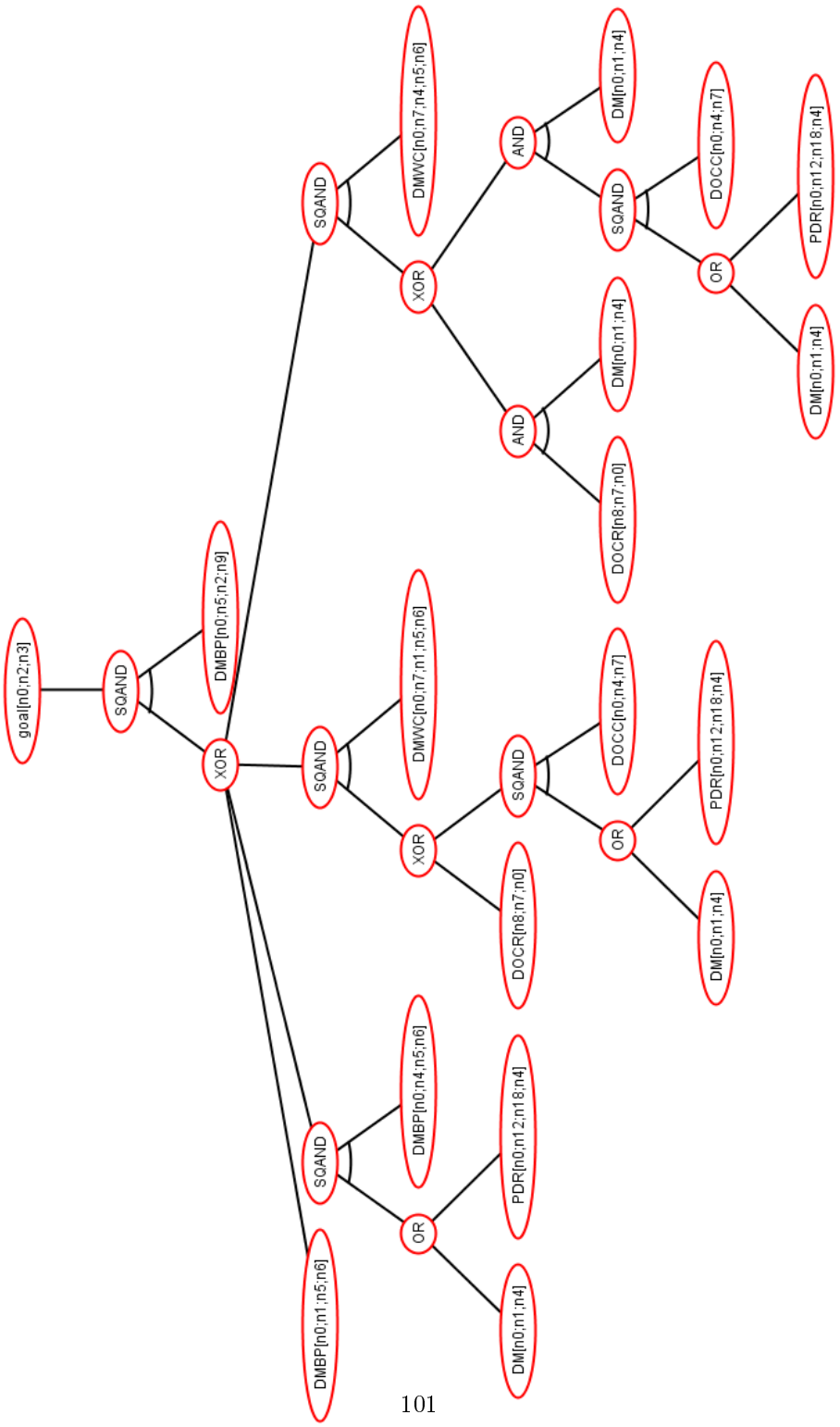


Figure A.2: Generated attack tree for a subset of 9 of the 40 identified attacks. Abbreviation of attacker actions used: DM = Digital move, DMBP = Digital move bypass policy, DMWC = digital move with credentials, DOCR = digital obtain credential from role, DOCC = Digital obtain credential from location, PDR = Physical digital relocate.

Bibliography

[1] Michael E Whitman. Enemy at the gate: threats to information security. *Communications of the ACM*, 46(8):91–95, 2003.

[2] TREsPASS. Technology-supported Risk Estimation by Predictive Assessment of Sociotechnical Security, FP7 project, grant agreement 318003. http://www.trespass-project.eu. 2012-2016. Online, accessed: 2015-08-11.

[3] VISPER. VISPER: The VIrtual Security PERimeter for digital, physical, and organisational security, project funded by the Sentinels programme. http://www.sentinels.nl/en/content/visper. 2007–2011. Online, accessed: 2016-01-25.

[4] SHIELDS. SHIELDS: Detecting known security vulnerabilities from within design and development tools, FP7 project, grant agreement 215995. http://www.shield-project.eu/en/pages/home/. 2008–2010. Online, accessed: 2016-01-25.

[5] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. Quantitative attack tree analysis via priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 156–171. Springer, 2015.

[6] Zaruhi Aslanyan and Flemming Nielson. Pareto efficient solutions of attack-defence trees. In *Principles of Security and Trust*, pages 95–114. Springer, 2015.

[7] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Attack–defense trees. *Journal of Logic and Computation*, page exs029, 2012.

[8] Aleksandr Lenin, Jan Willemson, and Dyan Permata Sari. Attacker profiling in quantitative security assessment based on attack trees. In *Secure IT Systems*, pages 199–212. Springer, 2014.

[9] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[10] Xinming Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345. ACM, 2006.

[11] Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. Towards synthesis of attack trees for supporting computer-aided risk analysis. In *Workshop on Formal Methods in the Development of Software (co-located with SEFM)*, 2014.

[12] Marieta Georgieva Ivanova, Christian W Probst, René Rydhof Hansen, and Florian Kammüller. Attack tree generation by policy invalidation. In *Information Security Theory and Practice*, pages 249–259. Springer, 2015.

[13] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. Automated generation of attack trees. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 337–350. IEEE, 2014.

[14] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM, 2002.

[15] Richard Paul Lippmann and Kyle William Ingols. An annotated review of past papers on attack graphs. Technical report, DTIC Document, 2005.

[16] Trajce Dimkov, Wolter Pieters, and Pieter Hartel. Portunes: representing attack scenarios spanning through the physical, digital and social domain. In *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, pages 112–129. Springer, 2010.

[17] Christian W Probst and René Rydhof Hansen. An extensible analysable system model. *Information security technical report*, 13(4):235–246, 2008.

[18] Zaruhi Aslanyan, Marieta G Ivanova, Flemming Nielson, and Christian W Probst. Modeling and analysing socio-technical systems. 2015.

[19] Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient unfolding of contextual petri nets. *Theoretical Computer Science*, 449:2–22, 2012.

[20] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using groove. *International journal on software tools for technology transfer*, 14(1):15–40, 2012.

[21] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review*, 13:1–38, 2014.

[22] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[23] Kenneth L McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification*, pages 164–177. Springer, 1993.

[24] Wolfgang Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[25] Leila Ribeiro. Parallel composition and unfolding semantics of graph grammars. 1996.

[26] Paolo Baldan, Andrea Corradini, Ugo Montanari, and Leila Ribeiro. Unfolding semantics of graph transformation. *Information and Computation*, 205(5):733–782, 2007.

[27] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[28] Javier Esparza and Keijo Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer Science & Business Media, 2008.

[29] Walter Vogler, Alex Semenov, and Alex Yakovlev. Unfolding and finite prefix for nets with read arcs. In *CONCUR'98 Concurrency Theory*, pages 501–516. Springer, 1998.

[30] Paolo Baldan, Andrea Corradini, and Ugo Montanari. An event structure semantics for p/t contextual nets: Asymmetric event structures. In *Foundations of Software Science and Computation Structures*, pages 63–80. Springer, 1998.

[31] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual petri nets, asymmetric event structures, and processes. *Information and Computation*, 171(1):1–49, 2001.

[32] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and event structure semantics for graph grammars. In *Foundations of Software Science and Computation Structures*, pages 73–89. Springer, 1999.

[33] Paolo Baldan, Barbara König, and Ingo Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *Graph Transformations*, pages 194–209. Springer, 2004.

[34] Paolo Baldan, Andrea Corradini, Barbara König, and Stefan Schwoon. Mcmillan's complete prefix for contextual nets. In *Transactions on Petri Nets and Other Models of Concurrency I*, pages 199–220. Springer, 2008.

[35] Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, 2008.

[36] Paolo Baldan, Andrea Corradini, and Barbara König. Unfolding graph transformation systems: Theory and applications to verification. In *Concurrency, Graphs and Models*, pages 16–36. Springer, 2008.

[37] Arend Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[38] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[39] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In *Information Security and Cryptology-ICISC 2005*, pages 186–198. Springer, 2005.

[40] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. Adtool: security analysis with attack–defense trees. In *Quantitative Evaluation of Systems*, pages 173–176. Springer, 2013.

[41] Wen-ping Lv and Wei-min Li. Space based information system security risk evaluation based on improved attack trees. In *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on*, pages 480–483. IEEE, 2011.

[42] Florian Arnold, Dennis Guck, Rajesh Kumar, and Mariële Stoelinga. Sequential and parallel attack tree modelling. In *Computer Safety, Reliability, and Security*, pages 291–299. Springer, 2015.

[43] Jin Bum Hong, Dong Seong Kim, and Tadao Takaoka. Scalable attack representation model using logic reduction techniques. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 404–411. IEEE, 2013.

[44] Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. Atsyra: An integrated environment for synthesizing attack trees. In *Second International Workshop on Graphical Models for Security (GraMSec'15) co-located with CSF'15*, 2015.