Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis

Converting Tamarin to extended Alice&Bob protocol specifications

Dorela Kozmai

dkozmai@student.ethz.ch

Supervisors: Dr. Ralf Sasse and Dr. Saša Radomirović
Professor: Prof. Dr. David Basin

Issue date: September 15th, 2015
Submission date: March 15th, 2016

Abstract

Alice&Bob notation is frequently used to describe security protocols, while protocol verification tools use their own protocol specification languages. One such protocol verification tool is TAMARIN. We are interested in a converter that translates a protocol specified in TAMARIN's protocol specification language to an extended Alice&Bob notation. The tool we develop takes TAMARIN security protocol theory `.spthy` files as input and produces the respective Alice&Bob `.anb` files. It is implemented in Python.

The approach we use has three main steps. We first parse the TAMARIN input and create an internal representation of all the building blocks of the protocol that are important for the conversion. TAMARIN's protocol specification language is based on rules. In the second step, we search for an executable sequence of these rules, by making use of an execution graph we create for each protocol. We simulate possible executions of the protocol by assigning values to variables in the rules. We use the tool Maude for unification modulo equations. Finally, in the third step, we use the executable sequence of rules and the respective assignments to create an Alice&Bob specification of the protocol.

Our tool correctly converts a subset of protocols that are created automatically by an already existing tool that converts Alice&Bob specifications to TAMARIN ones.

Contents

1 Introduction

When considering security protocols, proving that their behaviour is as expected for all scenarios, is essential to the security of our computer systems and the communication of information. Protocol verification tools, such as Proverif [1], Scyther [4] and Tamarin [6] are used for this purpose. These tools all use their own protocol specification languages. In this thesis we focus on Tamarin. A Tamarin protocol specification consists of a set of rules. Their order is arbitrary and they can be executed if their preconditions are met. This is very convenient for analyzing properties of the protocol, but it is neither very readable nor intuitive. Consequently, it is difficult to spot errors in a Tamarin specification.

An Alice and Bob, or A&B, text-book style specification is frequently used while discussing security protocols. Alice&Bob is an intuitive way of representing a protocol that resembles a narrator's description of a discussion between two or more communicating parties. We follow the extended Alice&Bob notation described in [2] and [5]. It goes beyond text-book style A&B and specifies the sender and receiver of each message, the messages that were sent, the necessary fresh values the senders need in order to send the messages, the knowledge the senders and receivers initially have, and the functions they are allowed to use in their interactions.

As intuitive as the A&B notation is, it is ambiguous and makes implicit assumptions. For example, it gives no information about how a receiver processes a message or what he does when the message does not have the expected form. The adversary behaviour is also not taken into consideration, unless if we are specifically explaining an intrusion scenario. Furthermore, it typically does not specify properties. If we were to use an Alice&Bob-like notation as our input language in a verification tool, we would have to define all these aspects through additional formalisms, which would change the language so much that we might as well introduce a new language that better fits our purposes of proving properties of the protocols. Thus, we use the previously mentioned verification tools.

Ideally we would want to be able to automatically switch between one specification and the other. This way we would get the best of what the two specifications have to offer. There already exists a tool for converting Alice&Bob specifications to Tamarin specifications [5], but not one that does the reverse. Therefore, in this thesis, we develop a converter of Tamarin specifications to extended Alice&Bob specifications.

2 Preliminaries

In order to explain how we convert TAMARIN specifications to Alice&Bob specifications, we first introduce these two protocol specification languages. These preliminaries are found in Section 2.1 and Section 2.2 respectively. Additionally, since the Maude tool [3] is used for an essential part in our project, we will also give a quick overview of Maude in Section 2.3. These overviews have a strong focus on the actual usage of the respective tools in our project and are not to be seen as general descriptive summaries of the tools.

2.1 Introduction to TAMARIN

TAMARIN is a security protocol verification tool. The inputs to the tool are the TAMARIN security protocol theory files, which are the relevant files for this project. They contain the protocol specifications, which are the specifications we want to convert to Alice&Bob specifications. Every such file consists of a theory. The theory is encapsulated between the `begin` and `end` keywords. A theory includes a signature specification, a protocol specification, and property specifications. The signature specification may include functions, built-ins and equations declarations, respectively signalized by the TAMARIN keywords `functions`, `builtins`, and `equations`. The protocol specification contains rules and axioms. The properties are expressed through lemmas.

We are interested in the protocol specification. Additionally, we need to also extract the signature specification, because it introduces the functions used in the protocol. The security properties of the protocol, however, are not our main interest in this project. One can express the properties one wants the Tamarin-prover to analyze in lemmas. They are necessary for the tool, whereas, to our conversion, the lemmas are irrelevant and we do not consider them or axioms in this project.

Definition 1. Let a and b be strings. We define the following representation for string concatenation with space: a `+` $b := a\ b$. We use the `'+'` symbol in infix notation. We define the following representation for string concatenation without space: $a * b := ab$. We use the `'*'` symbol in infix notation.

Definition 2. In the rest of this thesis, whenever we describe the form of a construct in a language, we put keywords and other symbols of the language inside quotation marks: `"keywords"`. All other terms that are used inside a description are to be seen as variables that can be replaced as explained in each case, and are represented in the description in italics: *`variables`*.

Terms and Facts

Definition 3. A term in a TAMARIN theory is either a variable, a function applied to terms or a constant function.

Definition 4. The sort structure of terms in TAMARIN.
Every term in TAMARIN is a message. Public messages and fresh messages are subsorts of the sort message.

A fact in a TAMARIN theory is of the form

`name` `+ "(" +` *`args`* `+ ")"`

where *name* is the fact name. A fact can have the property of being persistent, in which case *name* starts with the symbol '!'. Facts are either persistent or linear. A persistent fact can be consumed repeatedly, as opposed to linear facts, which can only be consumed once. *args* is a list of zero or more arguments. The arguments of a fact are terms. In a TAMARIN theory, if two facts have the same name, they must have the same number of arguments.

Definition 5. Let F be a fact in a TAMARIN theory. Then name(F) denotes the *name* part of F and args(F) denotes the *args* part of F.

Variables consist of only their names. There are different types of variables. Two examples are public variables, in which case their names may start with the symbol '$' and fresh variables, in which case their names may start with the symbol '~'.

The signature specification

Function declarations consist of the `functions` keyword, the ':' symbol, followed by a list of function names with their arities. These are represented in the form

name + "/" + *arity*

These are the functions that can be used to build terms. They are used inside the facts of rules and in the equations of the theory. The rules and equations are defined later in this Section. If a function is used in any fact of a rule or any term of an equation and it is not already an internally defined TAMARIN function, then it needs to be declared.

Equation declarations consist of the `equations` keyword, the ':' symbol, followed by a list of equations. An equation is of the form

$term_1$ + "=" + $term_2$

where $term_1$ and $term_2$ are both terms.

Built-ins can be words from the following list: `signing`, `symmetric-encryption`, `asymmetric-encryption`, `hashing`, `diffie-hellman`. When a built-in is declared, a list of corresponding equations and functions are additionally internally declared. Built-ins are declared following the `builtins` keyword and a ':' symbol.

The protocol specification

Rules are a TAMARIN construct that is very important for this thesis. They represent participants' actions. Rule declarations are of the form

"rule" + *name* + ":" + *L* + "[" + *left* + "]" + "-[" + *a* + "]->[" + *right* "]"

where `rule` is the TAMARIN keyword that signalizes a rule declaration. *name* is the name of the rule. *L* is either empty or it has the form

"let" + *replacements* + "in"

In this case, *replacements* consists of a list of bindings. These bindings are descriptions of syntactic replacements. Starting from bottom-up, one can replace everything in the left hand side of the bindings with the terms in the right hand sides in the *left*, *a* and *right* parts of the corresponding rule. The variable *a* is either empty or it contains facts, which are only needed for property specifications. Thus, we ignore this part of a rule. Finally, the `left` and `right` parts of the rule consist of lists of facts. They may be of arbitrary length.

There are two special facts in TAMARIN named `In` and `Out`. Their arguments represent input and output respectively. For a meaningful protocol the `In` facts must always be in the left part of a rule and the `Out` facts must be in the right side of a rule.

Definition 6. Let R be a rule in a TAMARIN theory. Then name(R) is the `name` part of R, let(R) is the `L` part of R, left(R) is the `left` part of R and right(R) is the `right` part of R.

Definition 7. The names of facts, rules, variables, and functions in TAMARIN can be chosen from a set of valid names. The set of valid names includes strings composed of letters, numbers, and the underscore symbol that start with a letter, or one of the following prefix symbols: $\{\$, !, \sim\}$. As long as they do not clash with the name of another construct, names can be chosen freely from this set of valid names.

The protocol execution

To understand how a security protocol can be specified with the previously mentioned constructs we need to give an intuition of how the rules in TAMARIN execute. As we have seen the rules have a left and a right side. We can see each rule as a mechanism that, given the facts in its left side, produces the facts in its right side. The scope of the variables used in each rule is local to the rule.

In order to be able to talk about the execution of a rule let us consider a set of facts, which we call general state. This state starts out empty and includes certain facts at a certain point in time. These are facts that are created by rules and are consumed by other rules. If they are not persistent they will be removed from the general state at the moment they are consumed.

The sending and receiving of messages in TAMARIN is achieved by the `In` and `Out` facts. TAMARIN follows a *Dolev-Yao* model. In such a model the adversary can hear and intercept every message. Additionally, if her knowledge allows it, she can synthesize the messages.

Since we are interested in Alice&Bob equivalent protocols, we can assume that when a message is in an `Out` fact, the adversary intercepted it and then put it back in the network. At this point the message can be received inside an `In` fact.

Consider the set of exchanged messages. Let us call it messages state. Here we have all messages that were at some point inside an `Out` fact in the right side of a rule that was executed. If a rule has an `In` fact in its left side, it will try to find one of these messages that matches the form of the term inside that fact. Only then will it have met its preconditions.

In order to specify a security protocol in TAMARIN we need to create certain rules, such that their execution is only possible in certain orders. The way we can do this is by setting up an initial identity fact for each participant in the protocol, which we will call

the initial state of the participant. We have a state for each participant, which can be thought of as his knowledge at the point of execution of a rule. We update this state after every rule execution and enforce in this way an order of execution for each set of rules that belong to a participant. We need to do this, since there are no restrictions on what order `In` and `Out` facts can be produced and consumed and we can not rely on any naming of messages. It is important to note that in the rule that created the message and in the rule that is consuming it, the message does not need to look the same to the participants. Intuitively we can think of it in the following way. The consuming rule has other facts in its left side, which could be thought of as its knowledge. The rule will thus see a message and try to express the message in the terms it knows. The other rule, which produced the message can have another knowledge set, thus has expressed the message in its terms.

Example 1. *Let `A` and `B` be protocol agents that have participated in the Diffie-Hellman key exchange protocol. In this protocol both participants agree on a value `g` and then create a fresh value each. They send `g` raised to the power of their fresh value to each other. Then they can raise the received value to the power of their fresh value and share this way a secret key. We observe them after this protocol run. `A` only knows `g`, `x`, which is her fresh value, and `v_b`, which is the value she received from `B`. `B` only knows `g`, `y`, which is his fresh value, and `v_a`, which is the value he received from `A`. Let us assume for the sake of the example that the participants now send the secret key in the public channel. `A` has the message `(v_b)^x` in the `Out` fact in her rule. In the messages state, this message will be the value of `g^(x*y)`. Say now `B` has the `In` fact in his rule telling him he needs the message `(v_a)^y`. `B` can in this case use the value `g^(x*y)` from the messages state.*

There exist TAMARIN specifications for which there are no equivalent Alice&Bob specifications. Examples of such specifications can be found in Section 3. We assume that for an Alice&Bob equivalent protocol the rules which are part of the interaction between the participants of the protocol are only executed once each per run of the protocol.

2.2 Introduction to extended Alice&Bob

Alice&Bob is a security protocol specification language. It provides an intuitive description of a communication event between two or more parties. Text-book Alice&Bob specifications do not include any property specifications. In this thesis we rely on the extended versions of Alice&Bob, as specified in [2] and [5]. From now on, when we use the term Alice&Bob, we mean the extended Alice&Bob. These specifications may include an informal `Goals` section. In this project we are interested in only translating the protocol specification, thus we do not discuss the goals of a protocol any further.

In an Alice&Bob specification of a protocol there is a fixed number of participants. These participants are all assumed to have an initial knowledge consisting of their own identity, public key, and secret key. For a given participant A, these are denoted as A, $pk(A)$, and $sk(A)$ respectively. In this thesis Alice&Bob is our output language. We use a different representation of the public key infrastructure. To each participant, we assign an identifier and another value used for their public and secret keys. Thus, the identity, the public key, and the secret key of a given participant A is represented as v_{1A}, $pk(v_{2A})$, and $sk(v_{2A})$. Participants may also have additional explicitly declared initial knowledge. In an Alice&Bob specification the knowledge of each participant is not explicitly declared in every step of their interaction. It is however assumed that participants expand their knowledge sets every time they create a new value or receive a message.

An Alice&Bob protocol specification may include a declarations section, a knowledge section and actions.

Definition 8. Terms in an Alice&Bob protocol specification can either be variables, numbers, participants' names, participants' knowledge, or functions applied to terms.

The declarations in an Alice&Bob protocol follow the `Declarations` keyword. Declarations serve to declare all functions that may be used in the protocol. A declaration is of the form

name + "/" + *arity* + ";"

Definition 9. Let D be a declaration in an Alice&Bob protocol. Then name(D) is the *name* part of D and arity(D) is the *arity* part of D. name(D) is the name of the function being declared and arity(D) is its arity.

The knowledge declarations in an Alice&Bob protocol follow the `Knowledge` keyword. They are of the form

P + ":" + *list* + ";"

where *P* specifies the name of the participant and *list* can be a list of terms that comprise the participant's initial knowledge.

The actions in an Alice&Bob protocol follow the `Actions` keyword. A sequence of actions describes a possible execution of the security protocol; generally, the ideal, successful scenario. They describe the communication between the participants of the protocol. An action is of the form

"[" + $name_{step}$ + "]" + A_1 + "->" + A_2 + *fresh* + ":" + *message* + ";"

Definition 10. Let A be an action in an Alice&Bob protocol. Then fresh(A) is the *fresh* part of A, message(A) is the *message* part of A, sender(A) is the A_1 part of A, receiver(A) is the A_2 part of A and tag(A) is the $name_{step}$ part of A.

In Alice&Bob, the names used for senders and receivers are identifiers and if a name is used multiple times in a protocol, it is referring to the same participant. However, our representation of public key infrastructure implies that the names of participants are not used inside messages. Instead their assigned identifiers are. In action A, fresh(A) includes all the new values that are freshly created by sender(A) of the message(A). There can also be actions were the *fresh* part is empty.

The messages consist of terms. The used variables are either created as a fresh value in the current or any of the previous actions, or are elements of the sender's knowledge set. The functions can either be functions that are declared in the declarations section or standard functions, such as hashing, symmetric encryption and decryption, asymmetric encryption and decryption, exponentiation and multiplication. Note that this list is not exhaustive.

The variable scopes in an Alice&Bob protocol specification are global, whereas in a Tamarin specification the scope of variables is local to each rule.

Example 2. *We are referring to Example 1. In the Alice&Bob specification,* `A` *would not send the message* `(v_b)^x` *to* `B`*. She would send a message similar to* `g^(x*y)`*. This is the case, regardless the actual representation of the message that* `A` *stores in her knowledge. Note that* `A` *still can not send or use the value* `y` *by itself at any point, because she does not know it.*

2.3 Introduction to Maude

Maude is a language and tool based on rewriting logic. One of its functions is formal verification of properties of a mathematical model. Maude is used by TAMARIN as a backend for unification modulo equations, which is done in TAMARIN's protocol analysis. In this project, we use Maude for its unification modulo equations as well. Note that Maude version 2.7 is needed.

The Maude module

A Maude module describes a model. The constructs of a Maude module that we need for our project are the sort, operator, equation and variable declarations.

A sort declaration is of the form

`"sort" + S + "."`

where `sort` is the Maude keyword denoting the beginning of a sort declaration. The `S` is the name of the sort. A sort defines a type. It is possible to define a subset of a sort using the `subsort` construct.

A variable declaration can be of the form

`"vars" + v + ":" + sort + "."`

where `vars` is the Maude keyword denoting the beginning of a variable declaration. The `v` part consists of the variable name. The `sort` part defines the type of the variable. It must be one of the sort names defined in the module. A variable must only be defined once in a module.

An operator declaration is of the form

`"op" + o + ":" +` $sort_1$ `+` $sort_2$ `+ ... +` $sort_n$ `+ "->" + sort + "."`

where `op` is the Maude keyword denoting the beginning of an operator declaration. The `o` part is the name of the operator. n is the arity of the operator. $sort_i$ is the sort of the i-th argument of the operator and `sort` is the sort the operator returns. An operator must only be defined once in a module.

One can specify properties of an operator, such as commutativity and associativity by including the keywords `[comm]` and `[assoc]` respectively in the operator's declaration.

Definition 11. A term in a Maude module is either a variable or an operator application.

An equation declaration is of the form

`"eq" +` $term_1$ `+ "=" +` $term_2$ `+ "."`

where `eq` is the Maude keyword denoting the beginning of an equation declaration. *term*$_1$ and *term*$_2$ are terms. All variables and operators used inside these terms must be declared in the same module. If an equation declaration has the `[variant]` keyword in it, the unification command takes that equation into consideration.

The `variant unify` command

The `variant unify` command in Maude is of the form

`"variant unify" +` *term*$_1$ `+ "=?" +` *term*$_2$ `+ "."`

where *term*$_1$ and *term*$_2$ are terms. The `variant unify` command is executed within the context of a module. All operators and variables used inside its terms must be defined in that module.

Maude tries to unify the two terms modulo the equations in the corresponding module. Given a supported equational theory, Maude computes a complete set of unifiers, if the two terms are unifiable.

3 Converting TAMARIN to extended Alice&Bob

For every Alice&Bob protocol specification there exists a TAMARIN protocol specification. But the other direction is not necessarily true. There exist TAMARIN specifications for which there are no equivalent Alice&Bob specifications.

Example 3. *Protocols with loops or branches can be specified in* TAMARIN*, but not in Alice&Bob.*

There are also other TAMARIN specifications for which there is more than one equivalent Alice and Bob specification. We are talking about different specifications, considering two specifications different, if they represent different protocols.

Example 4. *Consider a protocol where participant A sends out two different messages. Participants B and C receive these messages. Depending on which participant receives which message, we have two different protocol specifications.*

This is the case because protocols are specified through rules in TAMARIN and there is no fixed order of rules. Additionally, rules can be executed multiple times. In an Alice&Bob specification, there is one clear sequence of actions which specify the protocol.

Thus, for our conversion we need to find at least one possible executable sequence of rules in TAMARIN and calculate the equivalent Alice&Bob sequence of actions. Finding this executable sequence of rules is therefore not a trivial problem. To tackle this problem we create an execution graph, which acts like a guideline for our search for an executable sequence. We follow different heuristics in order to minimize our search domain. To check if a sequence of rules is executable, for each rule in the sequence we try to unify facts that are produced by previous rules with facts that need to be consumed by the current rule. For this unification we use Maude. Additionally, in a TAMARIN specification it is not necessarily clear who the participants in the protocol are, while in an Alice&Bob specification, they are stated clearly in every action. To extract the participants we use the execution graph and further heuristics.

The conversion from TAMARIN to the extended Alice&Bob protocol specification is done in three steps. In the first step the input is the input of the project, a TAMARIN theory file. We parse this TAMARIN input file and save the information relevant for the conversion in an internal representation. In the second step we use the internal representation to create an execution graph, test multiple execution sequences and check if they can be unified using Maude. At the end of this step our output is a successful sequence of rules and the information on their execution. In the last step we translate such an executable trace to an Alice&Bob specification and create the respective output file.

We now explain these steps in detail. In order to make this description more intuitive and clear, we choose a running example, the Diffie-Hellman key exchange protocol.

3.1 Parsing TAMARIN input and the internal representation

In this first step our input is a TAMARIN file and our desired output is the internal representation of the protocol constructs. In order to do this we use the two python modules, text_modifier.py and tamarin_parser.py.

Here we have the TAMARIN file for our Diffie-Hellman protocol. This is a TAMARIN file generated by the tool [5] that converts A&B specifications to TAMARIN specifications.

```
theory DIFFIE_HELLMAN
begin
functions: pk/1, sk/1, aenc/2, adec/2, g/0
builtins: diffie-hellman, symmetric-encryption
equations:
    adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1,
    adec(aenc(x.1, pk(x.2)), sk(x.2)) = x.1
rule Asymmetric_key_setup:
    [ Fr(~f) ] --> [ !Sk($A, sk(~f)), !Pk($A, pk(~f)) ]
rule Publish_public_keys:
    [ !Pk(A, pkA) ] --> [ Out(pkA) ]
rule Symmetric_key_setup:
    [ Fr(~symK) ] --> [ !Key($A, $B, ~symK) ]
rule Init_Knowledge:
    [ !Pk($A, pk(k_A)), !Pk($B, pk(k_B)), !Sk($A, sk(k_A)),
    !Sk($B, sk(k_B)) ]
    --[  ]->
    [ St_init_A($A, sk(k_A), pk(k_A)), St_init_B($B, sk(k_B),
    pk(k_B)) ]
// ROLE A
rule dh_1_A:
    [ St_init_A(A, sk(k_A), pk(k_A)), Fr(~x) ]
    --[  ]->
    [ Out((g() ^ ~x)), St_dh_1_A(A, ~x, sk(k_A), pk(k_A)) ]
rule dh_2_A:
    [ St_dh_1_A(A, x, sk(k_A), pk(k_A)), In(alpha) ]
    --[  ]->
    [ St_dh_2_A(A, x, sk(k_A), pk(k_A), alpha) ]
rule dh_3_A:
    [ St_dh_2_A(A, x, sk(k_A), pk(k_A), alpha), Fr(~n) ]
    --[ Secret_key_secret_A((alpha ^ x)),
    Secret_key_secretA_A((alpha ^ x)) ]->
    [ Out(senc{~n}(alpha ^ x)),
    St_dh_3_A(A, ~n, x, sk(k_A), pk(k_A), alpha) ]
// ROLE B
rule dh_1_B:
    [ St_init_B(B, sk(k_B), pk(k_B)), In(alpha) ]
    --[  ]->
    [ St_dh_1_B(B, sk(k_B), pk(k_B), alpha) ]
rule dh_2_B:
    [ St_dh_1_B(B, sk(k_B), pk(k_B), alpha), Fr(~y) ]
    --[  ]->
    [ Out((g() ^ ~y)), St_dh_2_B(B, ~y, sk(k_B), pk(k_B), alpha) ]
rule dh_3_B:
    [ St_dh_2_B(B, y, sk(k_B), pk(k_B), alpha),
    In(senc{n}(alpha ^ y)) ]
    --[ Secret_key_secret_B((alpha ^ y)),
    Secret_key_secretB_B((alpha ^ y)) ]->
    [ St_dh_3_B(B, n, y, sk(k_B), pk(k_B), alpha) ]
lemma key_secret:
    " not( Ex msg #i1 #i2 #j .
            Secret_key_secret_A(msg) @ #i1 &
            Secret_key_secret_B(msg) @ #i2 &
            K(msg) @ #j )"
lemma key_secretA:
    " not( Ex msg #i1 #j .
            Secret_key_secretA_A(msg) @ #i1 &
            K(msg) @ #j )"
```

```
lemma key_secretB:
    " not( Ex msg #i1 #j .
            Secret_key_secretB_B(msg) @ #i1 &
            K(msg) @ #j )"
end
```

The input to our program is such a TAMARIN file. We first modify this file in such a way that it is ready to be parsed easily.

3.1.1 Modifying the TAMARIN file

This is what the text_modifier.py does. First of all, it removes all single-line and multi-line comments. Comments in TAMARIN are C-style. '`/*`' denotes the beginning of a multi-line comment and '`*/`' the end. Multi-line comments allow bracketing. One line comments are denoted by '`//`' at their beginning.

The next task is to replace some TAMARIN constructs that would cause problems to our parsing algorithm, if they were not replaced. These are:
• the list representation: $< e1, e2, e3, ... >$
• the infix exponentiation representation
• the functions of the form: $func\{x1, x2, ...\}(y1)$.

Notation 1. *We use the symbol '$\longrightarrow$' in this Section to represent the transformation our modifier applies. This is not to be mistaken for the type of arrow in the rules of* TAMARIN*, the arrow in the actions of Alice&Bob, or the arrow in the operator definitions of Maude.*

Instead of a list, we create a function LIST of arity two. This way we can represent a list of one element as the element itself and a list of multiple elements as a LIST, where the first argument is the first element of the original list and the second argument is the function applied to the list containing all the elements except for the first.

Example 5. $< e1 > \longrightarrow e1$
$< e1, e2 > \longrightarrow LIST(e1, e2)$
$< e1, e2, e3 > \longrightarrow LIST(e1, LIST(e2, e3))$

We replace the infix representation of exponentiation with the prefix representation, creating the equally-named function ^ of arity two. The first argument of the function is the base and the second one is the exponent.

Example 6. $b\,\hat{}\,n \longrightarrow \hat{}(b, n)$

For the third problem, we replace the representation with one of a function with the exact same name, which takes two arguments, the first one being the output of the list replacing function with the list of the terms inside the curly brackets as an argument and the second one being the term inside the normal brackets. The list of functions we are supporting this representation for in this thesis is: `aenc`, `senc`, `sign`, `adec`, `sdec`.

Example 7. $aenc\{x1, x2\}(y1) \longrightarrow aenc(LIST(x1, x2), y1)$

Next we find all the `lemma` keywords and remove their declarations from our file, since they may include formal comments, which could cause problems to the parsing algorithm, because we do not parse this type of comment. As explained earlier, we are not considering the lemmas in this thesis. We mentioned that we do not consider axioms either. Axioms

can however be relevant for the protocol specification. In this thesis we remove them, too, but warn the user if a protocol specification contains axioms.

Finally we find the `end` keyword and remove everything that comes after it. We are now ready to parse the file. This is done by tamarin_parser.py. It takes the modified TAMARIN input file as input and produces an internal representation.

3.1.2 The internal representation

Let us first explain our internal representation mechanism. The parts we are interested in, in a TAMARIN file, when the task is to convert it to extended Alice&Bob specification are the functions, equations, built-ins and the rules, for which it is important to know the facts of their left and right sides. We also read the Theory name and the file name, but decide not to parse any other parts of a TAMARIN file.

To represent the previously mentioned constructs we create three types of objects in our Python implementation: Operators, Rules, and Equations.

Observation 1. The Operators in our internal representation can correspond to both facts and terms in TAMARIN. The Operators that are found in the array of Operators in either part of a Rule correspond to facts in TAMARIN. All Operators that are found in the arguments of Operators and either part of an Equation correspond to terms in TAMARIN.

An Operator has a name, which is a string and arguments, which are also Operators. The arguments correspond to the arguments of the fact or term in TAMARIN. One can easily get the arity of the Operator, as well as get and set its name and its arguments. Each Operator also has a multiplicity and a type. The multiplicity of an Operator is only relevant for Operators that correspond to facts in TAMARIN. It represents the number of times in a protocol that the fact appears on the left side of a rule. It is important to know the multiplicity of these Operators to determine the multiplicity of the Rules they belong to. This is described in more detail in Section 3.2.1. The type of the Operator is relevant for Operators that correspond to terms in TAMARIN. It can be a function with its arity or it can be *None*. This distinction is important when we need to distinguish between Operators that correspond to function applications and those that correspond to variables.

A Rule has a name, a left part, a right part, connections and a multiplicity. The name is a string and it corresponds to the `name(R)` of a TAMARIN rule R. The left and right parts are arrays of Operators and respectively correspond to `left(R)` and `right(R)`. The connections of a Rule are an array of Rules and the multiplicity is an integer. The definitions and the purpose of the connections and multiplicity can be found in Section 3.2.1.

An Equation only has a first and a second part, both of which are Operators. These correspond to the two terms of the equation in TAMARIN.

Now that we have explained what our input and output is, we can explain how we obtain the output from the input.

3.1.3 Parsing the modified TAMARIN file

First we parse the functions. These consist of all functions listed after the `functions` keywords. They are represented as a function name, followed by a slash, followed by the function's arity. Internally, the functions are represented as an array of tuples. The first

element of each tuple is the function's name and the second element of each tuple is the function's arity.

Next we parse the built-ins. These we save internally as an array of strings. After having looked up manually all the functions and equations that belong to a built-in, we hard-code what the program should do in case such a built-in is in the file. It adds all the functions to the existing function array and attaches a string consisting of the `equations` TAMARIN keyword and the listing of equations to the file. This way we can treat them as we would any other equations and we do not have to worry about the functions any more. Note that the `diffie-hellman` built-in is special. One can specify the equational theory of other built-ins in a TAMARIN theory file, but not that of `diffie-hellman`. In the case of the `diffie-hellman` built-in, we also need to additionally add an equation to account for the property:

$$(a^b)^c = a^{b*c}.$$

Except for adding this equation we of course also add the the function $*$ of arity two.

Next, we parse the equations. These consist of a left term, followed by an equal sign, followed by a right term. These terms are internally represented as Operators. In case the term is a function, its arguments are the Operator's arguments and its function name is the Operator's name. In case the term is a variable, the corresponding Operator has the same name as that variable and no arguments. Equations can be found in a listing after the `equations` keywords. Internally they are represented as an array of Equations, the first part of each Equation being the Operator corresponding to its left term and the second part being the Operator corresponding to its right term.

In order to parse the rules, we first have to add a new rule which is internally existent in TAMARIN and therefore not in our file. This rule is the Fresh rule. The Fresh rule is used to create new variables and has no preconditions. Thus it can always be executed. It is very important in an execution simulation, since as mentioned we start with an empty state. To account for the missing Fresh rule, we write a string that consists of all necessary parts of this rule and attach it to the file.

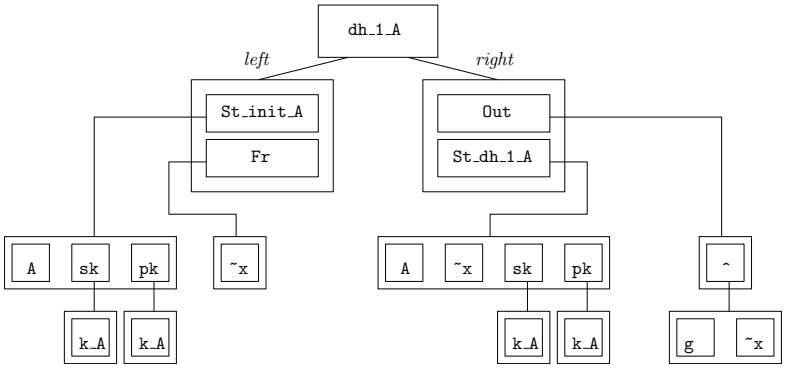
```
rule Fresh_rule:
[] --> [Fr(~x)]
```

Finally we parse the rules. To refer to parts of a rule we use Definition 6.

While parsing rule R, we first take care of the potential `let(R)`. We conduct the necessary replacements of terms in both `left(R)` and `right(R)`. Inside `left(R)` and `right(R)` there can be no, one or multiple facts. Thus we represent these facts internally as Operators. R is represented as a Rule with `name(R)` as the Rule's name, the arrays of Operators corresponding to the facts from `left(R)` and `right(R)` as left and right parts respectively and for the moment nothing as connections and 0 as multiplicity.

Figure 1 shows a visual representation of a Rule, where the Rule and the Operators are denoted by their names. In this rule there are two facts in the left side and two in the right side, which are represented by Operators. All Operators may have a connection to a list of other Operators. These are their arguments. Note that no Operator has an Operator that corresponds to a fact as an argument. If an Operator does not have any arguments it may correspond in TAMARIN to a variable, such as `k_A` or a unary function, such as `g`.

We now have the output of this program section and our internal representation: an array of rules, an array of functions and an array of equations. We have fully consumed




```
rule dh_1_A:
    [ St_init_A(A, sk(k_A), pk(k_A)), Fr(~x) ]
    --[  ]->
    [ Out((g() ^ ~x)), St_dh_1_A(A, ~x, sk(k_A), pk(k_A)) ]
```

Figure 1: The internal representation of the Rule corresponding to the rule `dh_1_A` from the Diffie-Hellman key exchange protocol

the input and from now on we only rely on our internal representation as the input for our next task: finding an executable sequence of the rules.

There is one last decision we make in this section. We delete all rules which only have an '`Out`'-Fact on the right hand side and only have persistent facts and/or Fresh facts on the left hand side. In our example the following is such a rule:

```
rule Publish_public_keys:
    [ !Pk(A, pkA) ] --> [ Out(pkA) ]
```

In general, these are rules which are not part of the interaction between the participants of the protocol, since those rules usually have either another fact in their right hand side to represent the participant's state at the moment of execution and/or a non-persistent fact in their left hand side, for the same reason. We do understand that this decision may affect the solutions for some protocols, but if we do not make it, we will get many orders as a solution and we will then have to use a heuristic. Since these rules that should not contribute to an Alice&Bob interaction may be the reason why we even find an executable ordering of the rules in such a case, we reckon that our chosen heuristic is safer to use. In the worst case, however, we are removing a rule that was relevant for the communication between parties and we fail to get a correct equivalent Alice&Bob specification. An example of such a case is the protocol where participant A only sends participant B her public key. One can write the Tamarin specification in such a way that our heuristic would cause us to remove the rules corresponding to this interaction. More about this decision can be found in Section 4.3.

3.2 Finding an executable sequence of rules

In this part of the project, our input is the internal representation of rules, equations and functions, that we created in the previous step. At the end of this step we want our

output to be a list of rules representing an executable sequence of the rules, a list of all facts consumed by rule executions, and a list of all facts produced by rule executions.

In order to achieve this, we first create an execution graph. This graph shows us which rules need to be executed before each given rule and how many times we need to execute each rule. It also gives us an idea about which rules correspond to different participants of the protocol and which rules are used for set up purposes. Then we try to run an execution of different possible sequences of rules. We test if a sequence is executable using Maude's unification function. As soon as we find one executable sequence of all rules we terminate this step. We acknowledge that there may be multiple executable sequences of rules, but we decide to only consider one.

We explain in detail how we create the graph in Section 3.2.1, how we find an executable sequence of all rules in Section 3.2.2, and how we use Maude for the unification in Section 3.2.3.

3.2.1 Creating the execution graph

This part of the solution is implemented in the module graph_maker.py.

We create a directed acyclic graph. Let us call the graph G, the set of nodes, `V(G)` and the set of edges `E(G)`.

The graph's nodes are Rules. A graph has a connection between $\texttt{R}_1 \in \texttt{V(G)}$ and $\texttt{R}_2 \in \texttt{V(G)}$ if there is an Operator in the left side of $\texttt{R}_2$ that has the same name and arity as an Operator in the right side of $\texttt{R}_1$ and if they unify.

Definition 12. Let $\texttt{R}_1, \texttt{R}_2 \in \texttt{V(G)}$.
$(\texttt{R}_1,\texttt{R}_2) \in \texttt{E(G)} \iff \exists\ \texttt{F} \in \texttt{left}(\texttt{R}_2),\ \texttt{F'} \in \texttt{right}(\texttt{R}_1)$ s.t. $\texttt{name(F)} = \texttt{name(F')}$, $\texttt{arity(F)} = \texttt{arity(F')}$ and the two Operators and their arguments are unifiable, as checked by Maude.

The graph shows us which rules we potentially need to have already executed in order to be able to execute a given rule.

We do not consider protocols for which the graph contains cycles any further. Generally, these protocols do not have equivalent Alice&Bob specifications.

We create `.dot` files for the graphs and then `.png` versions of them. Creating these graphs as an intermediate step is a good way of getting an idea of how the right execution order of the rules in the protocol looks like. Note that these graphs are created, even in the cases of protocols for which our program fails to produce an Alice&Bob specification.

An example of such a graph for the Diffie-Hellman key exchange protocol is shown in Figure 2. Note that in the graphical representation, we denote each node `R` by `name(R)` only, in order to make the graph understandable for a human viewer.

Calculating Rule multiplicities

Building the graph helps us in calculating the Rule connections and multiplicities.

Definition 13. Let R be a Rule. Then connections(R) is the set of all Rules that are children of node R in the execution graph.

The connections of a rule correspond to the rules which potentially consume facts produced by the given rule and thus are executed after the given rule. We are ignoring all `In`-s and `Out`-s in this step. We go through the Operators in the right side of a rule

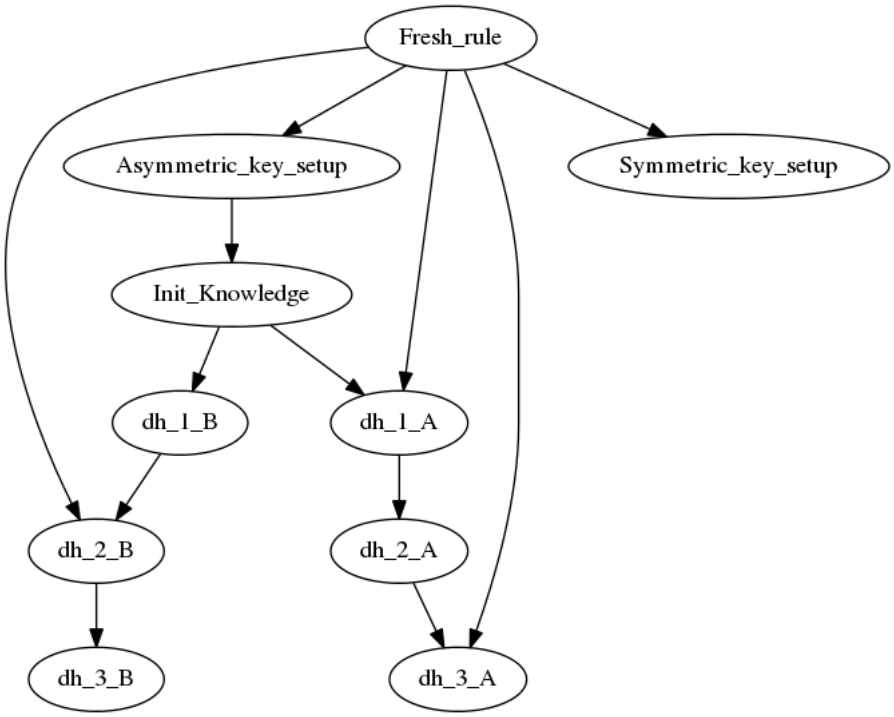


Figure 2: The execution graph of the Diffie-Hellman key exchange protocol

and then check that an Operator with the same name is on the left side of another Rule. However, we cannot be sure that we can connect the Rules yet. It could happen that there are two Operators with the same name and arity, but non-unifiable arguments, such as in Example 8. That is why in order to convince ourselves that the Rules should indeed be connected, we use Maude and try to find a unifier for the two Operators. If such a unifier exists we are content for now and without looking more into it, we add a connection.

Example 8. *Two* Tamarin *facts that have the same name and the same arity, but non-unifiable arguments:*

```
St_dh('1',<A, sk(k_A), pk(k_A)>)
St_dh('2',<B, sk(k_B), pk(k_B)>)
```

Strings are constants in Tamarin *and thus are represented as nullary operators, i.e., constant operators, in Maude. Therefore, the unification attempt for these two facts is rightfully unsuccessful in Maude.*

Calculating the multiplicity is more complicated, because of the nature of the rules. There can be multiple facts on each side of a rule and different facts produced by the same rule could be consumed by different rules or different executions of different rules. The multiplicity is supposed to tell us how many times a rule needs to be executed for the protocol specification to be executable. Finding the multiplicity is not trivial. We can easily calculate an upper bound. Since the multiplicity of a rule denotes the number of times the rule is executed in an execution simulation, using that upper bound increases the complexity of finding our solution a lot.

```
calculate_multiplicities_of_rules(G):
    for each rule R in V(G):
        multiplicity = calculate_multiplicity(R)
        if multiplicity is 0:
            multiplicity = 1
        multiplicity(R) = multiplicity


calculate_multiplicity(R):
    if connections(R) is empty:
        multiplicity = 0
    else:
        multiplicity = max([multiplicity(F) | F in second(R)])
        for connection C in connections(R):
            multiplicity_of_connection = calculate_multiplicity(C)
            if multiplicity_of_connection > 1:
                multiplicity += multiplicity_of_connection - 1
    return multiplicity
```

Figure 3: The algorithm for calculating multiplicities or rules

Thus, we are making the assumption that a rule that produces the facts $F_1,\ F_2,\ \ldots,\ F_n$ needs to be executed only as many times as the maximum number of rules having the same fact $F\ \in\ F_1,\ F_2,\ \ldots,\ F_n$ as a premise. In order to achieve this, we assign multiplicities to Operators that correspond to facts. As explained in Section 3.1.2, the multiplicities of Operators simply denote the number of rules in the TAMARIN specification that have the corresponding fact in their left side. By additionally using the Rules' connections, we can calculate the multiplicity of each rule. This is at least one, since even the rules that have no connections are potentially executed once. For other rules we first take the maximum multiplicity of its produced facts and then add to it the multiplicities of all the rules that are in its connections subtracting one each time.

Let us explain the intuition behind this algorithm. Let R be the rule we are calculating the multiplicity of. Let F be the fact with the highest multiplicity among the facts that R produces. Then R needs to be executed at least as many times as the multiplicity of F, since the multiplicity of F represents how many times F is consumed by other rules. Since we take the maximum we account for all rules that consume a fact produced in R being executed once. Additionally, we should take into consideration that each rule that consumes a fact that R produces has a multiplicity of its own. If such a rule R' has a multiplicity higher than one, which means it will be executed more than once, we execute R one less than multiplicity(R') more times too. The algorithm for calculating the multiplicities of rules is shown in Figure 3. Note: `connections(R)` includes every connection of rule `R` once.

These multiplicities of rules are in most cases the exact number of times the rule needs to be executed. They are an upper bound for rules that produce persistent facts and for different rules that produce the same fact. Having higher than needed multiplicities of rules does not cause a problem to executability, since the calculations are done bottom

up. Thus if we overestimated the multiplicity of a rule, the rules that need to be executed before it will also be executed more times than needed.

With the approach we use, it could happen that the multiplicity for a rule is underestimated. In example 9, we present a setting for which our multiplicity calculation would most probably hinder executability. We do not observe this setting very often in our TAMARIN protocol specifications and since it is very often the case that different facts that are produced by one rule get consumed by multiple rules, we decide to use our approach. We discuss the calculation of multiplicities of rules once more in Section 4.3.

Example 9. *Let R_1 be the rule that consumes fact F_1 and produces facts F_2 and F_3. Let R_2 be the rule that consumes fact F_2 and produces fact F_4. Let R_3 be the rule that consumes fact F_3 and produces fact F_5.*
$R_1 : \ F_1 \ \rightarrow \ F_2, \ F_3$
$R_2 : \ F_2 \ \rightarrow \ F_4$
$R_3 : \ F_3 \ \rightarrow \ F_5$
Let us assume that F_4 and F_5 are not consumed by any other rule. We calculate the multiplicities of these rules. The multiplicity of R_2 and the multiplicity of R_3 are both 1, since they have no connections. The multiplicity of R_1 would also be 1, since the multiplicities of its two produced facts are both 1. It could happen however, that rules R_2 and R_3 need instances of F_2 and F_3 that were created in two different executions of R_1. This means essentially that F_2 and F_3 are unifiable with the facts in the left side of R_2 and R_3 respectively, but need different unifiers. In such a case the multiplicity of R_1 should have been 2.

Finding the rules that belong to participants

We need to determine at some point which rules are part of the actual interaction between the participants of the protocol. Let us call these rules interaction rules. We additionally need to determine which rules belong to which participant. We do this based on information we extract from the execution graph.

First we decide which rules could potentially be interaction rules. Our first clue is that all these rules contain `In` or `Out` facts. Let us call the set of such rules relevant rules. Note that initialisation rules may also contain `Out` facts.

Next we cluster the relevant rules. We create a cluster for each participant. Since we do not consider `In`-s and `Out`-s in the graph, we expect rules that belong to different participants to be unconnected in the graph and rules that belong to the same participant to be. In case one or more of the relevant rules are initialisation rules, we expect to only get one cluster. In such a case we topologically sort the rules of the graph and take the relevant rule that comes first in the topological sort out of the set of relevant rules. We choose this heuristic, because initialisation rules generally need to be executed before interaction rules and thus come before interaction rules in a topological sort. Then we start the clustering procedure from the beginning. At the end we will have multiple clusters of rules and the union of all the rules in these clusters is the set of interaction rules. Note that for a protocol with only one participant this approach would cause problems. We do not consider this case any further, since such protocols are trivial.

3.2.2 Running a successful execution of the protocol

This part of the solution is implemented in the module graph_maker.py.

In this section we are trying to simulate a successful execution of the protocol. We tackle this problem in the following way. We test if different sequences of rules are executable. To determine the sequence, we try to guess the right order of execution of the rules and use our calculated multiplicity to define how many times a rule should be executed, deciding to perform these executions of the same rule consecutively.

For each given sequence, we start with an initially empty state, represented by an empty array. We always take the first rule from the sequence and we try to execute it. It will be executable if there exists a unifier of all the facts in its left hand side with existing facts in the state. Note that this is a 1-sided unification, which in principle is equivalent to matching. If a unifier is found, the facts that were in the state are removed from the state, if they are not persistent facts, and the facts that the rule produces are added to the state. If we manage to execute all rules in the order they are given in the sequence, then we had guessed a correct order and we continue with the conversion to Alice&Bob in the next section. We do not try to find all existing executable orders. If the attempt is unsuccessful, we try out another order.

Now let us describe this process in detail. We need to explain how we choose the sequences of rules and how we find a sequence that is executable by running a simulation of the rules in each sequence.

Choosing the sequences of rules

The execution graph we create gives us a first idea of how the execution order of the rules might look like. A topological sort of the graph is one such possible rule execution order.

Observation 2. We are talking about a rule order instead of a rule sequence because a topological sort does not tell us anything about how many times a rule needs to be executed. That is dealt with by our multiplicity calculations.

Definition 14. Let O_1 and O_2 be two different orders of rules. Then O_1 is equivalent to O_2 if in the execution simulations of both orders the same messages are exchanged between the same rules through `In` and `Out` facts.

For each order of rules we create one sequence of rules.

Definition 15. Let O be an ordering of rules. S is the corresponding sequence of rules, if all rules have the same relative order in S as in O, and each rule appears in S as many times as its multiplicity.

Thus for the order:
$O := R_1,\ R_2,\ \ldots,\ R_n,$
we get the sequence:
$$S := \underbrace{R_1,\ \ldots,\ R_1}_{multiplicity\ of\ R_1}\ ,\ \underbrace{R_2,\ \ldots,\ R_2}_{multiplicity\ of\ R_2}\ ,\ \ldots,\ \underbrace{R_n,\ \ldots,\ R_n}_{multiplicity\ of\ R_n}\ .$$

Next, we briefly argue why executing the same rule many times consecutively is a sound choice. We explained why we execute rules as many times as their multiplicity in the previous section. Since our graph is a directed acyclic graph, it cannot happen that different executions of two rules *need* to be interleaving. The alternative to our approach would be to only execute rules when they are needed. The important principle that both these approaches follow is that all facts that are needed for the execution of a rule need to have been produced beforehand by other rules.

Since we do not include the connections between `In`-s and `Out`-s in the graph, the topological sorts we get have the rules including them sorted only considering the other facts of the rules. Thus, our goal at this part is to find all possible topological sorts that could give us the right execution order, while keeping their number as small as possible.

We want to briefly argue why we use topological sorts.

Observation 3. In a successful execution of all rules we need the following property to hold: All rules that produce facts which are consumed by a given rule `R` in a successful execution, need to be executed before `R`.

The graph we have built has all the connections between a rule `R` and all the rules that produce facts that it consumes in a successful execution. The graph potentially includes other connections, too, but we are sure that the connections we are referring to in Observation 3 are a subset of the connections in the graph.

From the definition of a topological sort we know that in a topological sort of our graph the partial order of the rules is preserved. This way the rules that produce facts that could later be consumed by other rules come before them in a topological sort. So do, in particular, also the connections which are part of the successful execution. We conclude that a real successful execution needs to be a topological sort of the graph. Hence, considering only such orders saves us time.

Finding one topological order does not suffice. To make this clear, we take the following example of a topological sort of our Diffie-Hellman example graph.

Example 10. `[Fresh_rule, Asymmetric_key_setup, Symmetric_key_setup, Initial_Knowledge, dh_1_A, dh_2_A, dh_3_A, dh_1_B, dh_2_B, dh_3_B]`
A human can directly notice that this initial topological order is not the one we are looking for, because the rules corresponding to A and the ones corresponding to B are not interleaved.

At the same time, we do not want to find all possible topological sorts. Finding one topological sort of a graph is a task that can be solved in $O(n^2)$ steps. Finding all topological sorts, however, is a task that needs $O(n^n)$ steps in the worst case. Additionally, since the algorithm that tests executability of a sequence is costly on its own, especially when the order being tested is a wrong one, we want to test as few topological orders as possible. Luckily, we are able to derive all the topological sorts relevant to us if we have one initially calculated topological sort. Additionally we are able to judge to some extent how good a topological order is at this point.

We find an initial topological sort and then we make use of the knowledge on interaction rules. Since finding all topological sorts of a graph is a very hard and resource consuming task, we interleave the interaction rules and leave the rest of the rules in the same order as in the first discovered topological sort. Note that the rest of the rules are rules generally used for initialisation purposes. Thus, it is many times the case that we can permute these rules and get new topological sorts even if the interaction rules are left in the same order. These different sorts either all unify or none of them does. This is why we do not consider them any further.

The interleaving process can be imagined as follows: We find all permutations of interaction rules. For each permutation we take the rules in the permuted order and put them back in the places where the rules were found in the original topological sort.

Then we only consider the new orders that are topologically sorted. Example 11 shows all topological sorts we get from one initial topological sort for the Diffie-Hellman key exchange protocol.

Example 11. *From the execution graph of the Diffie-Hellman key exchange protocol in Figure 2 we can see that one initial topological sort of rules could be the following:*

```
[Fresh_rule, Asymmetric_key_setup, Symmetric_key_setup,
Initial_Knowledge, dh_1_A, dh_2_A, dh_3_A, dh_1_B,
dh_2_B, dh_3_B]
```

The interaction rules in this example are:

```
[dh_1_A,dh_2_A,dh_3_A,dh_1_B,dh_2_B,dh_3_B]
```

We now discuss the new orders of these rules that if placed in the initial topological sort give us a new topologically sorted order. These are interleavings of the two sequences: `[dh_1_A,dh_2_A,dh_3_A]` *and* `[dh_1_B,dh_2_B,dh_3_B]`*. The rules inside one sequence must remain in this relative order to one another in the final sequence. Thus, in this example we have* 6 *possible positions for rules and if we choose* 3 *of them for the rules that belong to A, there is only one correctly ordered sequence we get. Consequently, we get* $\binom{6}{3} = 20$ *different new orders.*

Considering only these orders instead of all topological sorts is a good start, but we can do better. We have already determined the rule clusters and we know if a given rule is a sending and/or receiving rule. A rule is sending if it contains an `Out` fact and receiving if it contains an `In` fact. We observe the ordered list of interaction rules. We decide to only consider orders for which for all pairs of consecutive rules in the ordered list interaction rules, if a rule that contains an `In` fact follows a rule that contains an `Out` fact, those two rules are in different clusters.

Example 12. *The list of orders that are still relevant after applying our new constraints in our Diffie-Hellman key exchange protocol.*

```
[dh_1_A,dh_1_B,dh_2_A,dh_2_B,dh_3_A,dh_3_B]
[dh_1_A,dh_1_B,dh_2_B,dh_2_A,dh_3_A,dh_3_B]
[dh_1_A,dh_1_B,dh_2_B,dh_2_A,dh_3_B,dh_3_A]
[dh_1_B,dh_1_A,dh_2_B,dh_2_A,dh_3_A,dh_3_B]
[dh_1_B,dh_1_A,dh_2_B,dh_2_A,dh_3_B,dh_3_A]
[dh_1_B,dh_2_B,dh_1_A,dh_3_B,dh_2_A,dh_3_A]
```

There could be an executable order of rules in which the above property does not apply. In such a case, there exists an equivalent order that respects the above property.

Simulating a successful execution of rules

Now we explain how our unification algorithm works. We create a state. This is an initially empty array that is later filled with produced facts. Note that all variables in these facts are assigned to constants. There will at no point be any fact in the state which contains an unassigned variable, public variables included. This is the case, because we are trying to

```
try_to_execute(sequence,state):
  if sequence is empty:
    executable = True
  else:
    first_rule = first rule in the sequence
    checked_possibilities = ∅
    while  ∃ possibility ∉ checked_possibilities:
      unifiable, updated_state =
         try_to_unify(first_rule,state,checked_possibilities)
      if unifiable:
        updated_sequence = sequence\first rule
        executable =
           try_to_execute(updated_sequence,updated_state)
      else:
        executable = False
        checked_possibilities =
           checked_possibilities ∪ current unification details
  return executable
```

Figure 4: The algorithm for the execution of a sequence in a state

run a simulation of an execution of rules. Thus, we assume that when a rule is executed, it produces concrete results.

Here is how we are testing if a sequence is executable. We start with all rules that have no facts in their left side. These are rules, such as the Fresh rule, which can create facts without needing to consume anything. We let them execute as many times as their multiplicity and for each execution we instantiate all created variables with new constants. These constants, the facts and the functions are converted to Maude notation. More about this can be found in the next section. We add the newly created facts in our state array. We want to stress that all facts in the state are in Maude notation. This first step is always done and is always the same for all the sequences we are testing. The rules that do not consume any facts can always be executed first, independent of where they appear in a sequence, since they do not depend on the execution of any other rules.

Then we take the next rule that appears in the sequence we are considering, and try to unify the facts in its left side with facts that are in the state array. If such a unification attempt is successful, we temporarily replace the consumed facts from the state array with the new facts that the rule produces, after having made the necessary substitutions of its variables. Then we recursively test that the rest of the rules can also be executed successfully in the given order. If this is the case, then we have found our order and we are done. We make the changes in the state definite. If the attempt fails, then we backtrack until we find a solution or until we can conclude that the sequence is simply not executable.

The algorithm is shown in Figure 4.

It can happen in a sequence that we manage to successfully execute one or more interaction rules and then fail to execute any further rule. In such a case we backtrack and try out different executions of previously executed rules. If in this backtracking procedure we reach an initialisation rule, we decide to stop our execution attempt in order

to get a shorter runtime. We observe that for most protocols if the initialisation rules are executed, and the execution fails at some later point, then it is an ordering problem of the interaction rules. Even if the initialisation rules were not executed as we want them to, for the right order of the interaction rules the protocol should be able to execute, possibly with wrong results. But if such an execution can not be run, then we conclude we have the wrong order of interaction rules.

If a sequence is not executable, we need to test if our next sequence is. This will only be the case if the next sequence does not start with the same $\mathtt{R}_{k1}$, $\mathtt{R}_{k2}$, $\mathtt{R}_{k3}$, ..., $\mathtt{R}_{kn}$ rules as any of the previously tested orders that failed, k being the order of the failed order, $_n$ being the order of the rule in an execution, for which the unification failed. We take the maximum depth of execution that was reached for the order. This way we do not waste time in testing if an order is executable, if we already know it is not. The maximum denotes the best possible execution attempt for a sequence. It may seem counter-intuitive, but if we do not take the maximum we risk ignoring sequences that might be executable. Let us explain this through an example.

Example 13. *Let us consider the sequence* R_1, R_2, R_3, R_4, R_5, R_6 *of* 6 *rules. Let us assume that in the execution attempt the unification once failed at rule* R_3 *and once at rule* R_4*. Note that the execution attempt indeed failed once at the third rule, but that does not mean that the rule is not executable at that order. The unification succeeded for that rule when it failed at the fourth. Consider sequence* R_1, R_2, R_3, R_5, R_4, R_6*. This may be executable, but any sequence that starts with* R_1, R_2, R_3, R_4 *is not.*

Next we explain how we determine if there exists a set of facts in the state that can unify successfully with the facts of the left hand side of a given rule. Our state array includes at any point in time all facts that were created until that point and not yet consumed by another rule, and all persistent facts that were created until that point. When we get a set of new facts, we try to find a subset of the facts in the state that can be used to instantiate them. If we could unify this subset of facts with the new facts in Maude, then we understand that the rule we are considering at this moment is indeed executable at this point in time. This means that the facts that we were able to use for the unification will be taken out of the state array and the facts in the right side of the considered rule will be added to the state array, after going through the variable substitution procedure. How we substitute in this case is different from the case of the rules with empty left sides, because in this case we need to use the result the Maude unification gives us for the variables that are used on the left side of the rule. Only if there are other new variables in the right side, will we apply the same substitution algorithm for them.

The algorithm is shown in Figure 5. For all the facts in the left side of the rule we first try to unify separately with facts in the state, every time using a fact from the state that is not yet used. In the case of persistent facts, we first try to unify with the fact that was not used before, but if that does not work allow the fact to be used twice, as is the case in a real execution. This is a heuristic we apply in order to avoid the instantiation of different public variables with the same name. Such public variables are used to represent the participants' identities and are often produced inside persistent facts. For this unification step we first check that the names of the facts are the same, and if that is the case we use Maude to unify the two facts. If we manage to unify all the facts in the left side of the rule with facts in the state separately, we use Maude to unify all the facts together. If it succeeds, we use the substitutions we derive from Maude's unifier on the facts on the right

```
try_to_unify(rule,state,checked_possibilities):
  facts = facts in left(rule)
  for f in facts:
    find fact f_state in state, s.t. name(f_state) = name(f)
    Maude unify f and f_state
    backtrack until (able to unify separately for all f-s in facts)
       and (the unification is not in checked_possibilities)
  Maude unify all facts together
  backtrack until able to unify all facts together
  if possible:
    unifiable = True
    apply Maude's unification in facts to the facts in right(rule)
    state = state ∪ facts in right(rule)
    for each fact that was unified from state:
       if it is not persistent:
        remove the fact from state
  else:
    unifiable = False
  return unifiable, state
```

Figure 5: The algorithm for the unification of facts in a rule with facts in the state

side of the rule. We then add these facts to the state array and remove all non-persistent consumed facts from the state array. We mark the used persistent facts. If no unifier can be found, we backtrack and try to find new separate unifiers for the facts in the left side of the rule.

Finally, we explain how we are substituting for the new variables. In Section 3.2.3, we mention that after parsing the Maude answer we get as an output a list of variables and their new assignments. We need to substitute these variables, that were matched in the left side of a rule, in the terms in the right side of the rule. We sort the list of substitutions in decreasing order according to the length of the terms that should be substituted. This way we avoid wrong substitutions of terms that have a name that is included in another term's name.

3.2.3 Using Maude

We use Maude both in the creation of the graph, as well as to find an executable sequence of rules. In both cases we use Maude to test if two or more facts are unifiable. For this we need to have a Maude module and a unification command. Note that there is only one Maude module per protocol, but we need to create many unification commands, since we want to unify facts of different rules and the unifications are dependent on previous unification results. Finally, we also need to take different actions for different types of Maude answers.

We use the python module maude_file_maker.py for all Maude-related purposes.

Creating the Maude module

What we have at this moment is our internal representation. From that we would like to generate a Maude module. As mentioned in the introduction, every operator or variable that is used in the unification command needs to be declared in the module. The same goes for functions and variables used in the equation declarations. Additionally, every type that is used in the declarations of variables and operators also needs to be declared as a sort in the module. So we first define our sorts. We observe that there is a distinction in TAMARIN between facts and functions, but none in Maude. In Maude both these constructs would be operators. In order to distinguish between them, we define a sort `F` for facts and a sort `M` for functions and variables. Note that functions and variables indeed are supposed to be of the same sort. Additionally we also define a subsort of `M`, `PVar` for public variables.

```
sort M .
sort F .
sort PVar .
subsort PVar < M .
```

Next we need to define operators. These correspond to facts and functions in TAMARIN. We use different prefixes while defining different Maude constructs. These are important for our conversion to the Alice&Bob specification language later. For now it is important that we define and use the constructs consistently in Maude. To find all facts we need to iterate through rules. For a rule `R` we check `left(R)` and `right(R)`. For each Operator `O` in `left(R)` and `right(R)` we define a Maude operator `o`
`op name(o) :` $\underbrace{M\ M\ \ldots\ M}_{\texttt{arity(O)}} \rightarrow \mathrm{F}$.
where we prefix the name with `"tamX"` to avoid name-clashes with Maude keywords:
`name(o) = tamX` $*$ `name(O)`

Example 14. *Consider the declaration of our example fact* `St_dh_1_A` *in Maude.*

```
op tamXSt-dh-1-A : M M M M -> F .
```

Note that the underscores have been replaced by dashes. This is a measure we need to take, since the underscore symbol is a special symbol in Maude. We understand that this may be an issue for protocol specifications where two constructs have the names `a_b` *and* `a-b`*.*

There is one case where we do not follow the above procedure and that is the case of `In`-s and `Out`-s. In both these cases the operator declaration is:

```
op tamXInOut : M -> F .
```

We need to keep in mind that there should be no duplicates of declarations. Thus we create a definition of an operator for a fact, the first time we come across the fact and ignore it in the future. Note that if there was a mistake and two facts with the same name appear twice in the TAMARIN file having two different arities, then in the unification attempt the number of arguments of one of them will not match the fact declaration signature, in which case the program exits with an error. Such a file is an invalid input for TAMARIN as well.

The other operators that need to be defined are the ones corresponding to functions. In TAMARIN, functions need to all be declared after the `functions` keywords or exist implicitly because of a built-in. There must be no functions that are used as fact arguments without being defined before. Thus we only go through the functions array to create the corresponding operator declarations. If a function is then illegally used in a fact without being declared, we exit with an error at the first occasion where this function appears in a unification attempt. For every function `fun` in our functions array, we declare the following operator in Maude:
`op name(o) :` $\underbrace{M\ M\ \ldots\ M}_{\text{arity(fun)}} \rightarrow$ `M .`
where we prefix the name with `"tamX"`:
`name(o) = tamX * name(fun)`

Example 15. *Two examples of operator declarations corresponding to functions from our Diffie-Hellman protocol:*

```
op tamXsenc : M M -> M .
op tamXg : -> M .
```

We mentioned before that in the case of the `diffie-hellman` built-in we additionally need to define a multiplication function and the equation which states the property: $(a^b)^c = a^{b*c}$. This has already been done. What we also need Maude to know is that it does not matter if we first calculate (a^b) and then raise it to the power of c or if we first calculate (a^c) and then raise it to the power of b. We cannot state this as an additional equation because it will cause an infinite loop in Maude. What we do instead is define the multiplication function as being commutative.

```
op prod : M M -> M [comm] .
```

Additionally, we use a special prefix for the list function, which is represented as a binary operator, and the string function, which is represented as a constant operator.

Next we declare our variables. These can be public variables, for which we use the sort `PVar` or non-public variables, for which we use the sort `M`. Variables can appear for the first time in equations or rules. In both these cases, in our internal representation, they will be Operators with no arguments, whose name is not in the list of functions. Thus we need to go through all equations and all rules and search for all variables. When we find one we declare it in Maude. Since in TAMARIN variables have a local scope, different rules or equations may use the same variable names. In our Maude module we want to acknowledge this and consistently rename the variables to be different. This is done by prefixing. The definition of a variable `v` in Maude for each variable `V` of our internal representation is of the form:
`vars name(v) : M .`
where we prefix the name with `"e"` or `"r"`, depending on if it is found in an equation or rule respectively, concatenated with the index of that equation or rule, and `"tamX"`:
`name(v) = e * index(E) * tamX * name(V)`, for non-public variable `V` inside the equation `E`, and
`name(v) = r * index(R) * tamX * name(V)` for non-public variable `V` inside the rule `R`.

For the public variables we follow the same procedure. The only part that is different is that instead of the sort `M`, they are declared to have the sort `PVar`.

Example 16. *Here is an example from our Diffie-Hellman key exchange protocol of the declarations of two variables of the same name appearing in two different equations.*

```
vars e0tamXx.1 : M .
vars e1tamXx.1 : M .
```

Finally we declare the equations. For that we need to go through all equations in our equations array and this time create an equation declaration that is very similar to the one that we parsed from Tamarin. The main difference is that here we use the newly defined terms as names of functions and variables. In the case of variables we create the names in exactly the same way as in the variable declarations. Everything that is not a variable but is an Operator inside the terms of an Equation is a function.

An equation declaration starts with the `eq` Maude keyword and for our purposes end with `[variant]`. This is necessary in order to be able to unify modulo the equations. If we do not add the `[variant]` description, the unification process will ignore the equations.

Example 17. *Two examples of equation declarations in Maude corresponding to equations from our Diffie-Hellman protocol:*

Tamarin *version:*

```
adec(aenc(x.1, sk(x.2)), pk(x.2)) = x.1
^(^(x_1, x_3), x_2) = ^(x_1, *(x_3,x_2))
```

Translated Maude version:

```
eq tamXadec(tamXaenc(e0tamXx.1,tamXsk(e0tamXx.2)),tamXpk(e0tamXx.2))
= e0tamXx.1 [variant] .
eq tamX^(tamX^(e4tamXx_1,e4tamXx_3),e4tamXx_2)
= tamX^(e4tamXx_1,prod(e4tamXx_3,e4tamXx_2)) [variant] .
```

Finally we discuss one last declaration we make in our Maude files. We define two operators `const` and `constP`. Both take a natural number as an argument and respectively return a construct of sort `M`, meaning a variable and a construct of sort `PVar`, meaning a public variable.

```
op const : Nat -> M .
op constP : Nat -> PVar .
```

We need such constants when we try to test if a rule sequence is executable. Whenever we apply a rule that has a variable in its right side, we want to assign this variable to a fixed value while keeping track of already set variables. This is the reason for these operators. Since we need to be able to use new values for new variables, we need to somehow remember what our already used names were, and having a counter is an efficient way to do that. Thus our operators take a natural number as an argument. In order to be able to use natural numbers in a Maude file though, we need to add the following declaration:

```
protecting NAT .
```

Creating the unification command

We create a unification command every time we call Maude for the unification of two terms. We follow two different approaches based on the nature of the terms we are trying to unify.

The first time we call Maude is while creating the execution graph. In this case we need to know if two terms are unifiable. Both terms have unassigned variables in them. Thus the procedure of rewriting these terms is the same as when we are declaring them. We use the same prefixes as in the declarations for facts, functions and variables.

The second time we use Maude is while trying to find a successful execution of a sequence of rules. In this case what we do is essentially matching. One of the terms we want to unify has no unassigned variables, while the other does. In this case the term with the assigned variables can be taken as it is, since it is in the desired format and is only using previously defined Maude operators and variables. This term is taken from the state and in the state all facts are written in Maude notation. For the other term we follow the same process as in the first case.

Finally, we explain how the unification command looks like for an attempt of unifying more than one pair of facts, which we do when trying to find a subset of the terms in the state that can be unified to all terms in the left side of a rule. In Section 2.3, we already explained how a unification command is constructed. Here it is once more:
`variant unify` $\texttt{term}_1$ =? $\texttt{term}_2$ `.`
If we need to unify n pairs of facts, it is constructed as a conjunction of queries in the following way:
`variant unify`
$\texttt{term}_{11}$ =? $\texttt{term}_{12}$ /\ $\texttt{term}_{21}$ =? $\texttt{term}_{22}$ /\ ... /\ $\texttt{term}_{n1}$ =? $\texttt{term}_{n2}$ `.`

Parsing the Maude answer

When using Maude in creating the execution graph we only want to know if two terms are unifiable. In that case if the Maude answer includes the string "No unifier", we conclude that there exists no unifier and we are done.

When using Maude for unifying two terms in the process of finding an executable sequence of rules, we need to parse the results inside a unifier and output a list of variables and their new assignments. Note that there can be multiple unifiers. We are parsing the information for all of them, but are in this thesis only considering the first one.

There are two different types of unifiers that Maude gives us. For some variables it unifies with a term where all variables are assigned. In such a case we directly add the variable and its new assignment in the output list. In the other case, Maude assigns variables to new variables in the following way. The `'%'` symbol followed by a number is the assignment. Its type, one of the predefined sorts, follows after the `':'` symbol. Note that this notation can be the whole assignment or it can be an argument of the assignment. In such a case we assign new `const`-s or `constP`-s, according to the type of each assigned variable. Finally, we add the original variable and its new assignment in the output list.

3.3 Producing an Alice&Bob specification

This part of the solution is implemented in the module tamarin_to_anb.py.

At this point in our project we have found an executable sequence of rules and we know what facts these rules consumed and produced while being executed. Additionally we have our graph, our participant clusters and our set of interaction rules as well as a list of all rules, functions and equations in our internal representation. From this information we need to produce an Alice&Bob specification. A complete specification consists of the protocol name, all necessary declarations, the initial knowledge of all parties, and the actions. We explain how we get the necessary information for each construct in Section 3.3.1. Note that all the produced and consumed facts are in Maude notation. Thus, so is in particular also the information we extract for the constructs. How we transform these terms into Alice&Bob suitable terms is explained in Section 3.3.2.

3.3.1 Extracting information for Alice&Bob constructs

We use the same name for the protocol as the name of the theory in the TAMARIN specification for consistency. In the case of declarations we decide to declare all functions that are in our function array in our internal representation. This way we are sure that we are not leaving any used function undeclared and since Alice&Bob ignores duplicates of function declarations, we allow these cases.

Example 18. *An example of such a case from our Diffie-Hellman protocol example is the declaration:* `senc/2`. *This is a predefined function in Alice&Bob for symmetric encryption..*

Next we determine the participants of the protocol and their initial knowledge. To do this we make use of our graph and the clusters. From the way we separate clusters, we know that rules from different clusters are not connected in the graph. We assume for this thesis that there is an initialisation rule that creates the initial states of all participants. We can find this rule in the graph in the following way. We search for a rule that has a connection to the first rule in each cluster. Recall that the rules inside one cluster are topologically sorted. Additionally, we make sure that the rule we find is not one of the rules that do not consume any fact, such as the Fresh rule. Finally, we create a new participant for each fact the rule produces. For the participant names we choose capital letters in increasing alphabetical order. We get the initial knowledge of the participant that was created for a given fact from the terms inside that fact.

Example 19. *In our Diffie-Hellman key exchange protocol the rule specification in* TAMARIN *of the rule that creates the initial states of all participants is the following:*

```
rule Init_Knowledge:
    [ !Pk($A, pk(k_A)), !Pk($B, pk(k_B)), !Sk($A, sk(k_A)),
    !Sk($B, sk(k_B)) ]
    --[  ]->
    [ St_init_A($A, sk(k_A), pk(k_A)), St_init_B($B, sk(k_B),
    pk(k_B)) ]
```

Thus, we create two participants named after the first two letters of the alphabet: `A` *and* `B`. *Their initial knowledge corresponds to the terms that are inside of the produced facts corresponding to* TAMARIN *fact* `St_init_A` *and* `St_init_B` *respectively. We get:*

```
A : v_0,sk(v_1),pk(v_1);
B : v_2,sk(v_3),pk(v_3);
```

Note that `v_0` *and* `v_2` *correspond to the identities of* `A` *and* `B`*. In Alice&Bob this knowledge is implicit. Every participant knows their identity and their public and secret key. We are not using those in our protocol specifications and use our defined terms instead.*

The next step is to assign one of these participant names to each cluster. We do this through comparing the consumed facts of the first rule of each cluster to the initial states mentioned above. We assign the same name to the cluster that includes a fact as to the participant that got the initial knowledge from that fact.

Next we describe how we get all the information we need for the actions. Let us refer to our definition of an action from our introduction.

"[" + $name_{step}$ + "]" + A_1 + "->" + A_2 + *fresh* + ":" + *message* + ";"

We instantiate the variables in an action as described below. The name of all actions is set to the name of the TAMARIN file and the step is increased for each created action. We can find the information we need for the messages `m` and fresh variables `f` by searching through the consumed and created facts. We go through the interaction rules and for each one that contains an `In` fact, we know that we should create an action. We can get the message in that action from the term that is inside the `In` fact. The fresh part consists of a list of terms that we can get from terms that are found inside `Fr` facts in the list of facts that were consumed by the rule. We mark this rule as a receiving rule. We additionally make sure that the number of `Out` facts contained in the interaction rules is the same as the number of `In` facts. However, for determining the messages, it is enough to only take in consideration either the terms that are inside the `In` or those that are inside the `Out` facts. We mark the rules containing the `Out` facts as sending rules. To determine the sender and receiver in each action, we find the cluster the sending and receiving rules belong to. The senders and receivers are the participants assigned to the clusters.

3.3.2 Generating the Alice&Bob notation

We have explained how we get the necessary information to instantiate for every construct in an Alice&Bob specification. We now explain how we transform the Maude terms into the terms we use. In our input all the facts that were consumed and produced are in our Maude notation.

The need for this translation is also a reason why we use the prefixes, such as `tamX` in our Maude modules. In this step we can use the knowledge we have about these prefixes to understand how we should transform each term. We are only interested in terms inside facts. The fact names do not matter for the Alice&Bob specification writing. Most functions have the `tamX` prefix. We remove that. That gives us the same function name that the function originally had in the TAMARIN specification. This is important, because in our declaration we take the function names directly from our internal representation, where they have those names, too. Special functions are the function that represents strings and the function that represents lists. For these two functions we use different prefixes. We then represent strings in A&B in the usual representation of a term encapsulated in between two `"'"` symbols. For lists we use the A&B notation of terms separated by a dot. For exponentiation, in order to keep the function in prefix notation and to avoid ambiguities, we rename it from `^` to `exp`. We also take care of the functions `aenc, senc, adec, sdec, sign`. We represent these functions in the following form: $func\{x1, x2, ...\}(y1)$.

Then we transform variables. These are represented as `const`-s and `constP`-s in Maude. We replace them with terms of the form `v_i` where i is a number. We keep track of all the mappings of the variables and for every new variable that we transform we increase i by one. We do not distinguish between public and non-public variables any more at this point, since in our Alice&Bob specification language there is no such distinction.

Finally, this is what our program outputs as an Alice&Bob specification for our Diffie-Hellman example protocol.

```
Protocol DIFFIE-HELLMAN:
Declarations:
pk/1, sk/1, aenc/2, adec/2, g/0, fst/1, snd/1, pair/2,
exp/2, sdec/2, senc/2;
Knowledge:
A : v_0,sk(v_1),pk(v_1);
B : v_2,sk(v_3),pk(v_3);
Actions:
[dh_0] A -> B (v_4): exp(g,v_4);
[dh_1] B -> A (v_5): exp(g,v_5);
[dh_2] A -> B (v_6): senc{v_6}exp(exp(g, v_5),v_4);
end
```

All the functions used inside the messages are declared and participants are either using values that are in their initial knowledge or freshly created ones. We also observe that no participant is sending any message that cannot be derived from their knowledge at that point. This specification describes the Diffie-Hellman key exchange with a confirmation message. We see that A sends g^{v_4} to B and B sends g^{v_5} to A. If B raises A's message to the power of v_4 and A raises B's message to the power of v_5, they both get the same value: $g^{v_4 * v_5}$. This is their shared secret key. In the last action, A sends the first message encrypted with the shared secret key.

This is the expected result. Recall that our Tamarin input was generated automatically by tool [5] from an original Alice&Bob specification, which is shown below for comparison. More about the differences between this original input and our generated output can be found in the next section.

```
Protocol DIFFIE_HELLMAN:
Declarations:
    g/0;
Actions:
    [dh_1] A -> B (x) : g()^x;
    [dh_2] B -> A (y) : g()^y;
    [dh_3] A -> B (n) : senc{n}(g()^(x*y));
end
```

Additionally, our generated specification is indeed easily readable, especially compared to the original Tamarin protocol specification.

4 Discussion

In this section we discuss some cases for which our tool can convert TAMARIN specifications to equivalent Alice&Bob ones, as well as discuss the difficulties we face in other cases.

4.1 Working examples

In this section we give examples of protocols converted by our tool. The TAMARIN specifications for these examples have all been created automatically with the tool [5] that converts Alice&Bob specifications to TAMARIN specifications, before our tool translates them back to Alice&Bob. Thus, we can compare our generated specifications with the original ones.

The first example is a signed version of the Diffie-Hellman key exchange protocol. We are presenting it, since it includes many different constructs, such as strings, lists, and the function for asymmetric encryption. In this example we can see how the participants used their initial knowledge in their messages.

```
Protocol SIGNED-DIFFIE-HELLMAN:
Declarations:
pk/1, sk/1, aenc/2, adec/2, g/0, fst/1, snd/1, pair/2, exp/2;
Knowledge:
A : v_0,v_1,sk(v_2),pk(v_2),pk(v_3);
B : v_0,v_1,sk(v_3),pk(v_2),pk(v_3);
Actions:
[signed_dh_0] A -> B (v_4):
                aenc{'One' . v_0 . v_1 . (exp(g,v_4))}sk(v_2);
[signed_dh_1] B -> A (v_5):
                aenc{'Two' . v_1 . v_0 . (exp(g,v_5))}sk(v_3);
end
```

This is the original A&B specification, before it was converted to TAMARIN. We have omitted the `Goals` section.

```
Protocol SIGNED_DIFFIE_HELLMAN:
Declarations:
    g/0;
Knowledge:
    A: pk(B), B;
    B: pk(A), A;
Actions:
    [dh_1] A -> B (x) : aenc{'One' . A . B . g()^x}sk(A);
    [dh_2] B -> A (y) : aenc{'Two' . B . A . g()^y}sk(B);
end
```

What we notice is that in our specification, we have more declarations. This is because some of them are not being used in the messages and some are implicitly declared functions. The initial knowledge of the participants seems to also be greater in our specification, but this is because the identity, public key and private key of each participant is implicitly known in the original Alice&Bob specification. As mentioned in Section 2.2, we choose a

different representation for our public key infrastructure. Thus we need to specify the name the participant is known with, and the value they use for their public and private keys. We observe that both variables v_0 and v_2 correspond to A in the original specification. The same goes for v_1 and v_3, and B. The exponentiation function is also an implicitly declared function that is represented as the '^' symbol in infix notation. In our specification we declare the function exp explicitly and use it in prefix notation. This was also our choice.

The next example is the Alice&Bob representation of the CR protocol, a challenge-response protocol.

```
Protocol CR:
Declarations:
pk/1, sk/1, aenc/2, adec/2, fst/1, h/1, pair/2, snd/1;
Knowledge:
A : v_0,sk(v_1),pk(v_1),pk(v_2);
B : v_3,sk(v_2),pk(v_2);
Actions:
[cr_0] A -> B (v_4): aenc{v_4}pk(v_2);
[cr_1] B -> A : h(v_4);
end
```

Here is the original protocol specification in Alice&Bob notation, before it was converted to Tamarin. We observe the same differences as in the previous example.

```
Protocol CR:
Knowledge:
    C: pk(R);
Actions:
    [cr1] C -> R (n) : aenc{n}pk(R);
    [cr2] R -> C     : h(n);
end
```

Finally, we have a longer protocol, the ASW fair exchange protocol for contract signing. It uses asymmetric encryptions and hashing.

```
Protocol ASW:
Declarations:
pk/1, sk/1, aenc/2, adec/2, fst/1, h/1, pair/2, snd/1;
Knowledge:
A : v_0,v_1,v_2,sk(v_3),pk(v_3),pk(v_4);
B : v_0,v_1,sk(v_4),pk(v_3),pk(v_4);
Actions:
[asw_0] A -> B (v_5): aenc{pk(v_3) . pk(v_4) . v_2 . h(v_5)}sk(v_3);
[asw_1] B -> A (v_6):
aenc{aenc{pk(v_3) . pk(v_4) . v_2 . h(v_5)}sk(v_3) . h(v_6)}sk(v_4);
[asw_2] A -> B : v_5;
[asw_3] B -> A : v_6;
end
```

The original A&B protocol is shown below.

```
Protocol ASW:
Knowledge:
    A : m, pk(B), B;
    B : pk(A), A;
Actions:
    [asw1] A -> B (n_1) : aenc{ pk(A) . pk(B) . m . h(n_1) }sk(A);
    [asw2] B -> A (n_2) :
    aenc{ aenc{pk(A) . pk(B) . m . h(n_1)}sk(A) . h(n_2) }sk(B);
    [asw3] A -> B : n_1;
    [asw4] B -> A : n_2;
end
```

We again observe the same differences as in the previous examples.

4.2 Restrictions posed by the solution

There is one main issue that causes our conversion attempt to fail for certain protocols. In these cases, an executable sequence of rules cannot be found. This happens because of the assignment of public variables. While testing the executability of a sequence, when using Maude to unify facts on the left side of a rule with facts that are in the state, we assign all public variables that are created from a rule to new constants. Thus it happens that a public variable that represents one participant is assigned to two different constants. This causes the unification to fail in rules where two facts that include these supposedly equal variables are found. The issue will become clear in the example below.

Unfortunately, this is the case for the TAMARIN specifications similar to the ones that can be found in the Tamarin-prover examples directory. Let us call these specifications "the usual specifications" in this section, as opposed to the "automatic specifications", which we call the ones that are automatically generated by the Alice&Bob to TAMARIN tool [5]. There are several differences between these two types of specifications.

There exists a rule that creates the initial states of all participants in the automatic specifications, and each participant takes that initial knowledge with him in each rule, modifying it when needed. In the usual specifications, the participants are not given any explicit initial knowledge, nor do they keep track of their knowledge at a given point. They can multiple times "forget" their identities and keys and they look them up whenever they need them in a rule.

We can see this clearly in the execution graphs of these rules. In Figure 2, Section 3.2.1 we have the graph for the Diffie-Hellman key exchange protocol. Its TAMARIN specification is an automatic specification. We can see how in this graph the node that represents the `Init_Knowledge` rule has two children, such that the descendants of one child have no connections to the descendants of the other child. The path of each child and its descendants represent the rules that correspond to each participant. Some of them are connected to the `Fresh_rule`, but none of them is connected to other initialisation rules. In contrast to that we have in Figure 6 the graph for the NSLPK3 protocol as specified in the examples directory of the Tamarin-prover. We see the paths of rules in this graph that correspond to each participant, but all these rules have connections to the initialisation rules, and not only to the `Fresh_rule`.

The problem with the public variables is caused when two rules create a public variable that is supposed to represent the same identity and the public variable is assigned to a

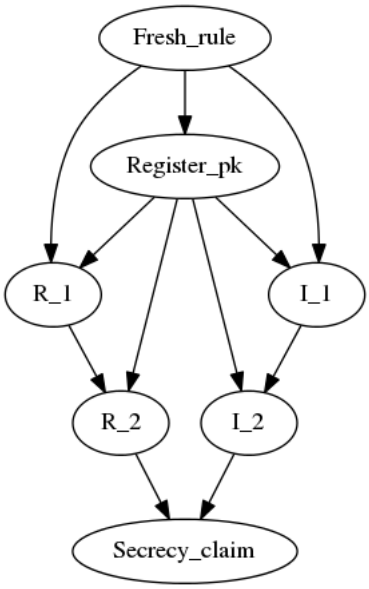


Figure 6: The execution graph for the NSLPK3 protocol, for the specification in the Tamarin-prover examples directory

constant.

Example 20. *Let us examine how this happens in the example specification of the NSLPK3 protocol from the Tamarin-prover examples directory. We show only the two rules that cause a problem when executed.*

```
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA))], Out(pk(~ltkA)) ]

rule I_1:
  let m1 = aenc{'1', ~ni, $I}pkR
  in
    [ Fr(~ni)
    , !Pk($R, pkR)
    ]
  --[ OUT_I_1(m1)
    ]->
    [ Out( m1 )
    , St_I_1($I, $R, ~ni)
    ]
```

The rule `Register_pk` *assigns in one of its executions a constant to variable* `$A`*, that is supposed to represent the identity of participant* `I`*. We see however that in rule* `I_1`*, the variable* `$I`*, that is also supposed to represent the identity of participant* `I`*, appears in the right side of the rule first. Thus it is not bound and will be assigned to a new*

constant. Further in the execution attempt of other rules it will be impossible to unify the two different constants to the same term.

One can try to solve this issue the following way. One knows how many participants the protocol has and can decide to only create that many new public variables. This would be a possible heuristic, since the public variables are often only used to represent the participant identities. Then one would need to keep track of all other choices one makes for the other used public variables. To give an idea of the complexity of this solution, let us consider a two party protocol where there are n appearances of public variables. The first two would be assigned to constants. We need to consider that these first two might need to be the same one, but let us assume we do find the first two different ones. Then for the remaining $n-2$ appearances we have every time two options. Consequently, in the worst case, the procedure needs $O(2^n)$ steps. We cannot avoid this, as long as we are using forward substitution for our unification.

Additionally, we want to point out that for the usual specifications one needs to follow a different procedure in order to determine the initial knowledge of each participant. We suggest this is also done in the future. A way to do it would be to set the initial knowledge of a participant to the union of the terms inside the facts that it consumes and produces, removing all the terms inside the `Fr` and `In` facts.

We also suggest a different approach on the division of rule clusters for the usual specifications. Our approach on this matter is explained in Section 3.2.1. In such specifications frequently all rules that contain `In` and `Out` facts are connected. Our approach works for certain protocols, but it relies on the fact that initialisation rules come before the interaction rules in a topological sort and this is not necessarily the case for such protocols.

4.3 Future work

In this section we present suggestions to improve various aspects of the tool. We already mentioned three of them in the previous section. Those are very important changes, since they would expand the set of protocols that can be converted. Other important discussion topics are presented below.

The rule multiplicities we calculate can be essential to the executability of a protocol. We mentioned before that we do not calculate the upper bound for the multiplicities, but try to estimate the exact number of times a rule needs to be executed for a successful run. As the presented example shows us, this could in some cases be underestimated. We choose our approach because of the added complexity of finding a unifiable execution of rules that is caused by setting multiplicities to their upper bound. What one could alternatively do, is calculate the upper bound and then only execute the rules as many times as our calculated multiplicity. If no solution is found, one could increase the multiplicities as long as they do not exceed the upper bound. How the multiplicities are increased remains to be decided. For executable protocols where the upper bound is an overestimation and our multiplicities are large enough, finding the solution would take the same amount of steps as it does now. For non-executable protocols, we would be certain that the multiplicity was not the reason why the simulation failed, but the complexity of the solution would increase.

Another decision that could cause problems for certain protocols is taking out the rules that only have an `Out` fact in the right side and only have persistent and/or Fresh facts in

the left side. The problem arises because of the difficulty of automatically differentiating between `Out` facts that are necessary for the participant interaction and the `Out` facts that are there only to build the knowledge of the adversary. Finding a solution for this problem is suggested as future work.

One could also decide to find multiple solutions for one protocol and then analyze them. This would help understand where the solution we find fits in the bigger picture.

Our last remark is on trying to make the solution faster. The part of the solution that takes the longest to compute is the attempt to execute rule sequences, especially non-executable ones. One could make use of parallel computing techniques, since the computation for a sequence is not dependent on that of other sequences. Other ways of making the program faster include trade-offs. One has to apply certain heuristics. We suggest applying a different heuristic than the current one for the multiplicities of persistent facts. These facts generally need to have lower multiplicities, since they can be consumed more than once.

5 Conclusion

In this project, our goal was to build a tool that converts TAMARIN specifications of security protocols to Alice&Bob. For a security protocol it is important to know what messages are exchanged between the participants of the protocol. In TAMARIN these messages are inside `In` and `Out` facts. One also needs to know which participant sends and receives each message. This in TAMARIN is not explicit. There are however certain facts that appear in the same rules that have `In` and `Out` facts, that seem to include a participant's knowledge. These are all parts of the protocol one needs to extract in order to generate an Alice&Bob specification. We followed a systematic approach and clearly specified the assumptions we made. First, we extracted every information that is necessary to understand a TAMARIN specification out of the input file. Then we tried to find an executable sequence of all rules. The simulation of a successful execution of all rules, gives one all the needed information on the participants and messages. However, there are many possible rule sequences, so we needed to get some more information about rule executions. In order to find one executable sequence of all rules, we first created an execution graph. This graph shows all the possible dependencies each rule has on other rules and gives a partial order of rules. Additionally, the graph provides a way to distinguish different participants in a protocol. We calculated the number of participants from the graph and determined which rules belong to which participant. We generated sequences that are most likely to be executable and then run simulations of executions of these sequences of rules. After finding an executable sequence, we continued with the final step. We created the Alice&Bob protocol specification. While running the simulation, the rules that were executed consumed and produced facts. These facts include all the information needed to generate the output A&B files.

Our tool can successfully convert certain protocol specifications, but fails to convert others. There is a problem with the way we test if a sequence is executable. Every time we execute a rule, it consumes facts and produces other facts. We assign all new variables in the newly produced facts to new constants, public variables included. This causes problems for certain protocol specifications, if in the simulation of an execution more values are assigned than needed. We strongly suggest a different approach for this step as the first improvement to our project.

The tool that converts TAMARIN to Alice&Bob protocol specifications and this thesis is available at `http://www.infsec.ethz.ch/research/software/tamarin.html`. The tool's input is an `.spthy` file. One can run the tool with the command: `"tamarin_to_anb tamarin_file.spthy"`. The tool's output is an `.anb` file that includes the Alice&Bob equivalent protocol specification, as well as a `.png` file that shows the execution graph of the protocol.

References

[1] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *ProVerif 1.90: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2015. Originally appeared as Bruno Blanchet and Ben Smyth (2011) ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.

[2] Carlos Caleiro, Luca Viganò, and David A. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Cas J.F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. *Proceedings of the 15th ACM conference on Computer and communications security - CCS 08*, 2008.

[5] Michel Keller. Converting Alice&Bob Protocol Specifications to Tamarin, Bachelor's Thesis, ETH Zurich, 2014. Available at `http://www.infsec.ethz.ch/research/software/tamarin.html`.

[6] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. *2012 IEEE 25th Computer Security Foundations Symposium*, Jun 2012.