Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Model Checking Security Protocols with ProB

Masterarbeit

vorgelegt von

Miles Vella

21. Juli 2025

im Studiengang Informatik
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Dr. Jens Bendisposto

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 21. Juli 2025

Miles Vella

Abstract

The goal of this work is to develop a domain-specific language (DSL) to represent security protocols and the properties that they should uphold in the Dolev-Yao formal model, and then translate the language into a formal specification that can be animated, visualized and model checked with the ProB toolchain. Security protocols, which are communication protocols involving cryptographic primitives, are ubiquitous in today's world and due to the architecture of today's traffic a security flaw in a simple protocol deep down in the network stack may have far-reaching consequences. Even established protocols like the Needham-Schroeder protocol from 1978 can have security problems, as shown by Gavin Lowe in 1995, 17 years later, with model checking.

This work presents the Dolev-Yao formal model for security protocols and the security properties that can be verified with it. Upon this formal model a concrete DSL using *lisb* is built, so that – starting from a single representation – multiple different formal specifications can be generated, for example classical B, Event-B and Prolog via XTL. It is demonstrated how the ProB toolchain can be used to process those generated specifications and how the ProB toolchain was improved to make all of this feasible. Then some implemented techniques are explained that allow exploring the whole state-space and improve the performance of model checking. Finally, a case study and benchmarks are performed to compare the different output formats to each other and to existing protocol verification tools, and to show that explicit state model checking is a feasible technique for the automatic verification of security protocols, with some caveats.

Acknowledgements

I would like to thank Michael Leuschel for his support with ProB and always being willing to help solve problems, Philipp Körner for the fruitful discussions and for building *lisb* which made this work possible and Jan Gruteser for the helpful discussions about TLA$^+$ and implementing TLC support into ProB2-UI.

Contents

1 Introduction

In today's world, communication between digital devices in distributed systems or on the internet is essential to their function. Safety-critical and private information is transferred via insecure channels to establish communications, deposit money or control machinery. But what makes any network insecure is the possible presence of an adversary or intruder that listens to or even modifies the network traffic. Therefore, strategies must be developed to safeguard all important data and to maintain the intended functionality of all connected devices.

Communication protocols govern how the participating agents talk to each other - who sends what messages and when. They are called *security protocols* (or alternatively *cryptographic protocols*) when they involve cryptographic operations like encryption or hashing. Description and specification of communication protocols used in real networks differs based on the assumed abstraction level and the required capabilities of the underlying transport protocols, for example the TCP/IP stack used by the modern Internet [For89].

In this work, security protocols operate on the basis of messages being transmitted from a sender to a receiver. Normally, it is assumed that the underlying network guarantees that they will not get lost in transit, unless an adversary wills it, or switch their ordering. It is not required to manually add low-level sender or receiver address information. The protocol contains a sequential list of instructions to build, send and receive messages using cryptographic primitives like encryption, signing and hashing. Correctly designed security protocols will safely transfer secrets and ensure the identity of all parties, but this correctness is not self-evident. So we need techniques to validate whether a given protocol fulfills its declared properties.

Examples of commonly used protocols are SSL/TLS [Res18], which is used for most encrypted internet traffic in the form of HTTPS, SSH [Ylo06], which is used for remote login and control in the Unix world, IKEv2 [Kau+14], which is used to secure VPN traffic as part of IPsec, and Kerberos [Neu+05], which is used for authentication in Windows.

The goal of this work is to automatically verify the correctness of security protocols using the ProB toolchain and the B method. The rest of this thesis is structured as follows: first we will take a look at other tools for the automatic verification of security protocols to differentiate this work, then the background and the formal Dolev-Yao model for security protocols will be presented, then a domain-specific language will be developed based on the formal model, after that a tool will be presented to translate that language to different formalisms supported by the ProB toolchain, namely B, Event-B and XTL machines, and finally the generated machines will be evaluated by comparing them to each other and to existing tools.

2 Related work

Before defining the formal Dolev-Yao model, we look at existing tools which allow automatic verification of security protocols.

ProVerif *ProVerif* [Bla01] is a command-line tool that allows for the verification of security protocols with a Dolev-Yao adversary, given a representation in Pi calculus or rewrite rules. Internally it translates the protocol to Horn clauses, which are also used by Prolog. The tool can deal with unbounded sessions and supports symmetric and asymmetric encryption, hashing and Diffie-Hellman key agreements. The checkable properties include secrecy, authentication and equivalence properties. It can only provide these features because it is making some approximations which may lead to wrong traces of counter-examples being produced, but if it claims that a protocol fulfills a property, this property is indeed a true statement. Additionally, there is no way to animate a protocol or to step through a trace, one run of the tool will provide console output or a bare HTML page which includes possible counter-examples. Both input languages require some boilerplate before actually implementing the protocol and the properties have to be formulated manually. Furthermore, the protocol description language is hard to write and understand because it has little in common with the formal Dolev-Yao model.

Scyther *Scyther* [Cre08] is another tool for the automatic verification of security protocols with a Dolev-Yao adversary. The input language is intuitive and follows the formal model closely because it uses a free term algebra to model messages and the different processes are modeled independently of each other. This limits the boilerplate in comparison to *ProVerif*. The built-in properties that can be checked are secrecy and authentication. But additionally there is the possibility to categorize the protocol, yielding a finite representation of possible behaviors. *Scyther* comes as a graphical user interface which allows editing the protocol description and running the verifications directly and showing the results and possible counter-examples. There seems to be no way to initiate a verification from the command-line and the editor lacks features like syntax highlighting. And although the graphical visualizations are very clear and structured, it is not possible to step through a trace or to manually set up a protocol run.

Tamarin *Tamarin* [Mei+13] is the successor of Scyther, being more powerful and more similar to a full mathematical prover. It has a command-line interface and an interactive graphical interface locally accessible via a web browser. To model a protocol, *Tamarin* uses multiset rewriting rules, and the implementation of a protocol requires a lot of boilerplate again, and all protocol properties have to be manually specified using first-order logic. *Tamarin* uses symbolic proofs and thus can deal with unbounded process counts. After verifying a property it will output a proof if possible, or a counter-example if one was found, but because of the abstract nature of the protocol specification is difficult to mentally

map them to concrete traces. Although the graphical interface allows interactive proving, there is no way to step through a trace or to manually set up a concrete protocol run. Furthermore, the protocol description language is hard to write and understand because it has little in common with the formal Dolev-Yao model. Messages are not sent and received, but instead facts are defined and queried that represent those concepts.

AVISPA The *AVISPA* [Vig06] project is a whole toolchain for the automatic validation of security protocols. It specifies an input language called HLPSL and provides multiple backends for checking. There is also the third-party animator interface *SPAN* [Glo+06]. The input language HLPSL is a verbose variant of the Dolev-Yao formal model, where each role has to be described in its own block with the local state modeled explicitly. Shared data and the adversary knowledge is modeled by the implicit `environment` role. Properties are built-in and are listed in a separate block. *AVISPA* provides an intermediate representation, which is produced by the front-end and then passed to multiple backends which use different techniques for verification. The different backends include a SAT translation and a model checker.

Notably, none of these use explicit state model checking and the options for visualizing attacks and protocol execution are poor. That is the gap we are trying to fill with this work. The goal is to develop a protocol description language based on the Dolev-Yao formal model, that is as easy to use as *Scyther*'s, can be used as an input for multiple backends like *AVISPA*, but has more support for graphical visualization and manual protocol execution than comparable tools. Using ProB and the B method allows us to easily support multiple different formalisms and model checkers. Instead of reinventing the wheel and developing a new language for security protocols with a parser and analysis pipeline, we make use of *lisb* to develop a simple domain-specific language based on Clojure syntax.

3 Background

3.1 The B method

The B method [Abr96] is a formal method to specify and verify software components using the B language which uses set theory and first-order logic to describe those components as a state-based transition system. This language is called classical B, to differentiate it from the later developed Event-B [Abr10].

The B method has been successfully applied in the industry, especially in the railway sector [But+20]. An example of this is the driverless Paris metro line 14, for which all safety-critical components have been designed and validated with the B method. Additionally, the B method can be used for data validation [But+20; HSL16]. In this use case, data is loaded into a B machine and then checked for correctness with a number of predicates. It can also be used to model security and communication protocols, as demonstrated by Abrial [Abr96] and by Benaissa & Méry [BM10]. But those models were manually designed, in this work we want to focus on automatic verification instead.

3.2 Classical B and Event-B

Both classical B and Event-B are designed similarly. They are mathematical languages that are strongly typed, each identifier must have a type assigned to it by a predicate. A modeled component is called a *machine*, and it has a set of *variables* describing its state and a set of state-independent *constants*. The values of constants are constrained by a predicate called *property* or *axiom* in classical B and Event-B respectively. State variables are constrained using the *invariant*, which is a predicate that must be true in all reachable states, else there is an error. State transitions can be accomplished using *operations* or *events* respectively. They are defined with a variable amount of *parameters* and a *guard* which specifies when the transition can be executed and possible values for the parameters, and finally a *substitution*, which are instructions for changing the current state, for example by assigning a new value to a variable. A more in-depth look into the differences of both formalisms was done by Leuschel [Leu21]. In both variants, classical B and Event-B, one can describe systems and components, use refinement to transform an abstract representation to a more concrete representation that can be used to generate an executable program, apply model checking techniques and formulate rigorous mathematical proofs about system properties. An example of a classical B machine can be seen in Listing 1.

For working with Event-B there exists Rodin [Abr+06], an integrated development environment for the Event-B language and its proofs. It is based on the Eclipse platform and can be extended with plugins. Proofs are a central part of the Rodin workflow and the *correct by construction* mantra. Rodin generates a lot of proof obligations that need to pass for the model to count as correct. For example the invariant preservation proof obligation, which is generated for every invariant and every event. It needs to prove that

Listing 1: A classical B machine modeling a counter

```
1:  MACHINE Counter
2:  VARIABLES x
3:  INVARIANT x : NATURAL
4:  INITIALISATION x := 0
5:  OPERATIONS
6:      inc = x := x+1;
7:      inc_by(d) = SELECT d : 1..3 THEN
8:          x := x+d
9:      END;
10:     dec = SELECT x > 0 THEN
11:         x := x-1
12:     END
13: END
```

the invariant is true after executing an event, given that the invariant was true before. This is needed to prove via induction that the invariant is true in every reachable state.

3.3 ProB

ProB [LB03; LB08] is an animator, constraint solver and model checker for the B method, that is written in SICStus Prolog [CM11]. It can process classical B, Event-B, XTL [LM00] specifications written in Prolog and TLA$^+$ [Lam99] modules, another formal specification language.

ProB extends the B language with additional ways to create new algebraic datatypes, which are records and freetypes. Records are similar to structs from other programming languages and work like named tuples, each component has a name and a type. In a type theoretic way they are equivalent to tuples, as they are just ordered list of types, and are an example of product types. Freetypes are an advanced feature which are derived from Z [PL07], an earlier notation for formal specifications originally developed by Abrial [Abr74; ISO02], which is where the name *free type* comes from, and the Event-B theory plugin [BM13], which allows the definition of such data types. They constitute sum types and are similar to tagged unions or variants in other languages: a value of a freetype can take one of a finite set of cases and may include some payload data specific to that case.

ProB itself provides two user interfaces and a command-line interface for writing B models and for interacting with them. The older and simpler editor is called the ProB Tcl/Tk interface, and the newer editor written in Java [Gos+25] on top of the ProB Java API [Kör+20] is called ProB2-UI [Ben+21]. Both of these editors are optimized for animation and model checking, as opposed to the Rodin editor, which is more focused on proofs, but ProB provides a plugin to implement these features into Rodin.

Animation allows the user to manually verify the model's behavior by repeatedly selecting an outgoing transition from a list of possible transitions, while observing the current state of the model, and thus exploring the state-space. The ProB toolchain also supports visualizations, which is the displaying of the current state or the path to the current state as a graphical image.

3.4 TLA$^+$

TLA$^+$ [Lam99], short for *Temporal Logic of Actions*, is one of the formalisms supported by ProB. It is a formal specification language designed by Leslie Lamport and not part of the B method, even though it is quite similar. With the B method it shares the notion of a state with variables and static constants, transitions that change the state, and the semantic which is based on sets and first-order logic. An example of a TLA$^+$ specification can be seen in Listing 2.

TLA$^+$ has its own model checker called TLC [YML99] that is implemented in Java. Thus, a translation between TLA$^+$ and classical B is feasible, and was implemented by Hansen first from TLA$^+$ to classical B [HL12] and then from classical B to TLA$^+$ [HL14], making model checking B specifications with TLC and model checking TLA$^+$ modules with ProB possible. The TLC model checker exhibits better performance for certain setups than the native ProB model checker, so it will be used as an additional option where possible.

Listing 2: A TLA$^+$ module modeling a counter

```
1:  ---------------------- MODULE Counter -------------------------
2:  EXTENDS Naturals
3:  VARIABLE x
4:  ---------------------------------------------------------------
5:  Init == x = 0
6:  Inc == x' = x + 1
7:  IncBy(d) == x' = x + d
8:  Dec == x > 0 /\ x' = x - 1
9:  Next  ==  Inc \/ \E d \in 1..3: IncBy(d) \/ Dec
10: Spec == Init /\ [][Next]_x
11: ===============================================================
```

3.5 Clojure

Automatically generating B code or interacting with the ProB Java API to create an abstract syntax tree of a B machine is tedious and error-prone, thus we will use *lisb*, a library for programmatically interacting with B code written in Clojure.

Clojure [Hic20] is a modern LISP [McC60] dialect that runs on the Java Virtual Ma-

chine [Lin+25]. It shares many of its properties with other languages from the LISP family: it is a functional language that is written with s-expressions and the linked-list is a fundamental part of its syntax and semantic. Clojure features even more immutable data structures like the *map*, an associative container, the *vector* which is a sequence that supports random access and the *set* which is an unordered collection of unique values. Another inherited feature is the REPL (*read-eval-print-line*), a mechanism to execute code on the fly and to rapidly prototype functionality. Due to its homoiconicity, code is written as native data structures and can be processed as such, Clojure has a very strong and well integrated macro system. Defining a macro is as simple as defining a function, and they are passed the unevaluated code structures, which can be manipulated like any other data structure in the language. Clojure has an extensive standard library and can use *host calls* to access libraries on the Java Virtual Machine, which allows seamless integration with the Java ecosystem, especially the ProB Java API [Kör+20].

An example of a Clojure program containing three different functions for calculating the n-th Fibonacci number is shown in Listing 3.

Listing 3: The Clojure programming language

```
1:  (defn fib-naive [n]
2:    (if (< n 2)
3:      1
4:      (+ (fib-naive (- n 1))
5:         (fib-naive (- n 2)))))
6:
7:  (defn fib-loop-recur [n]
8:    (loop [n n
9:           fib-n 1
10:          fib-n-minus1 0]
11:     (if (< n 1)
12:       fib-n
13:       (recur (dec n)
14:              (+ fib-n fib-n-minus1)
15:              fib-n))))
16:
17: (defn fib-seq
18:   ([]
19:     (fib-seq 1 0))
20:   ([fib-n fib-n-minus1]
21:     (lazy-seq (cons fib-n
22:                     (fib-seq (+ fib-n fib-n-minus1)
23:                              fib-n)))))
```

3.6 EDN

A subset of Clojure, called EDN (Extensible Data Notation) [Hic12], can be used as a serializable data format to exchange Clojure data structures. This is similar to JSON [Bra17], which was designed as a subset of JavaScript [Ecm25] for data exchange. An example of a data structure encoded with EDN, that shows the most important features, is presented in Listing 4. It is very easy to read an EDN document into Clojure-native data structures with the built-in `read` function. Thus choosing EDN as an input format saves work by not having to develop an additional parser.

Listing 4: The extensible data notation (EDN)

```
;; semicolon marks a comment
;; braces {} delimit an associative map, and contain key-value pairs
;; all containers may be heterogeneous
;; commas count as whitespace, so they may be added for readability
{:keyword :foo, ;; colon marks a keyword, they are similar to enumerations
 :null-element nil, ;; equivalent to null in Java
 :boolean true, ;; and false
 :integers 42, ;; positive and negative integers
 :floats 1.337, ;; floating point numbers
 :strings "foobar", ;; string literals work like they do in Java
 :characters \c, ;; character literals always start with a backslash
 :symbol foobar, ;; symbols represent identifierse
 :lists (1 true :foo), ;; parentheses mark lists
 :vector [1.0 bar "s"], ;; brackets mark vectors, which support random access
 :sets #{false \newline [1 2]} ;; hash-prefixed braces mark mathematical sets
}
```

3.7 *lisb*

Due to the B language being rather inaccessible to programmatic analysis and generation, Körner & Mager developed *lisb* [KM22; KMR25] to rectify these shortcomings. *lisb* implements its own intermediate representation, which can be created in Clojure code or translated from an existing B model. Additionally, a DSL for the specification of sequential algorithms is implemented as a case study. The intermediate representation can then be exported as a classical B machine for immediate use. Furthermore, translation into Event-B models was implemented by Armbrüster and Körner [AK24].

lisb takes an EDN representation of a machine, an example can be seen in Listing 5, as input and then internally generates an abstract syntax tree (AST) with the ProB Java API for a classical B or Event-B model and then outputs a printed version, which looks like Listing 1. We will use *lisb* to easily generate classical B and Event-B machines from the same EDN representation, instead of manually building an AST for each or concatenating

strings.

Listing 5: Counter model in *lisb* notation

```
(machine :Counter
  (variables :x)
  (invariants
    (member? :x natural-set))
  (init
    (assign :x 0))
  (operations
    (:inc [] (assign :x (+ :x 1)))
    (:dec [] (select (> :x 0) (assign :x (- :x 1))))))
```

4 The Dolev-Yao adversary model

This chapter will present the Dolev-Yao adversary model [DEK82; DY83] based on the work by Basin, Cremers & Meadows [BCM18]. The following definitions are taken from their work, with slight alterations.

The Dolev-Yao adversary model is a formal model which is used to describe security protocols, the context they run in, the possible actions of an adversary and thus provides a way to mathematically prove their properties.

The model is very a high-level abstraction, the minutiae of networking, like bytes, headers and routing are not relevant. The network as a distinct entity is not modeled, the adversary takes the role of a messenger instead. During normal operations we can assume that no messages are ever lost or are transmitted out of order, and that message contents do not change. But, as the adversary has full control over all messages being sent and received, they may drop messages, create new messages and alter messages to execute an attack.

4.1 Basic terminology

The Dolev-Yao model is term-based, that means each message that is sent between the participating agents is a term. But terms themselves are not only atomic, they may recursively contain other terms, which are called subterms. An agent assumes a role in a protocol and executes a sequence of steps. They may build a term to send to the other party or receive a term and destructure it.

To get to a formal definition of the model, we start with the basic building blocks: terms.

Definition 4.1 (Basic sets)**.** *We define the following pairwise-disjoint infinite sets:*

- Agent*, the set of all agents*
- Role*, the set of all roles in the protocol*
- Fresh*, the set of freshly generated terms, also called nonces*[1]
- Var*, the set of all variable names*
- Func*, the set of all function names*
- TID*, the set of all thread identifiers*

We also define AdvConst $\subset$ Func*, the set of all adversary generated constants, which they may use as a replacement for* Fresh *values when generating message terms.*

[1] A cryptographic nonce is an arbitrary number that is only used once. So it must be freshly generated for each protocol run.

Definition 4.2 (Basic terms)**.** *With the above,* BasicTerm *can be defined as containing:*

$$\begin{aligned}\mathsf{BasicTerm} := \quad & \mathsf{Agent} \\ & \cup \mathsf{Role} \\ & \cup \mathsf{Fresh} \\ & \cup \mathsf{Var} \\ & \cup \mathsf{AdvConst} \\ & \cup \{x\#\mathrm{tid} \mid x \in \mathsf{Fresh} \wedge \mathrm{tid} \in \mathsf{TID}\} \\ & \cup \{x\#\mathrm{tid} \mid x \in \mathsf{Var} \wedge \mathrm{tid} \in \mathsf{TID}\}\end{aligned}$$

Where the suffix $\#\mathrm{tid}$ *specifies localized terms, that are owned by a specific thread given by the identifier* tid*.*

Definition 4.3 (Recursive terms)**.** *New terms can also be built from existing terms. Let* Term *be the smallest set satisfying the following predicates:*

$$\begin{aligned}x \in \mathsf{BasicTerm} &\implies x \in \mathsf{Term} && \\ x, y \in \mathsf{Term} &\implies (x, y) \in \mathsf{Term} && \textit{a pair} \\ x \in \mathsf{Agent} \cup \mathsf{Role} &\implies \mathrm{pk}(x) \in \mathsf{Term} && \textit{the public key of } x \\ x \in \mathsf{Agent} \cup \mathsf{Role} &\implies \mathrm{sk}(x) \in \mathsf{Term} && \textit{the secret key of } x \\ x, y \in \mathsf{Agent} \cup \mathsf{Role} &\implies \mathrm{k}(x, y) \in \mathsf{Term} && \textit{the shared key between } x \textit{ and } y \\ \mathrm{msg}, \mathrm{key} \in \mathsf{Term} &\implies \{\mathrm{msg}\}_{\mathrm{key}} \in \mathsf{Term} && \textit{the encryption of } \mathrm{msg} \textit{ with } \mathrm{key} \\ f \in \mathsf{Func} \wedge x_1, \ldots, x_n \in \mathsf{Term} &\implies f(x_1, \ldots, x_n) \in \mathsf{Term} && \textit{call of } f \textit{ with arguments } x_1, \ldots, x_n\end{aligned}$$

Function calls may be used to model constants (with $n = 0$*) or hash functions*[2]*.*

The term algebra must be able to model cryptographic operations as well. The most important of these is encryption, of which there are two types: symmetric-key and asymmetric (also known as public-key). For symmetric-key encryption, one shares a key with another party and then uses plaintext data and the shared key with a fixed encryption algorithm to produce ciphertext which can be shared safely, as only the intended recipient can decrypt it using the same shared key. The asymmetric-key encryption generates a key pair instead of a single shared key. One is the so-called public key which may be shared with the public, and the other one is the secret key which must stay private at all times. When encrypting some plaintext with this method, one needs to either use the public or the secret key as an additional input to the encryption algorithm, but decryption is only possible if the other key is known. So if the public key was used for encryption, the receiver of the ciphertext must use the secret key to decrypt it. And on the other hand if the secret key was used for encryption, the public key must be used for decryption, which is a safe way to prove that the sender has actually written the contained plaintext.

[2]Hash functions are functions which map input data into some fixed output space, where it is impossible or at least highly improbable to find the preimage of a given hash value.

There are many concrete algorithms for both categories of encryption. For example, symmetric encryption has AES [NIS23] and ChaCha20 [Ber08], which are based on XOR transformations, while asymmetric encryption has RSA [RSA78] and elliptic curves [Kob87; Mil86], which are based on the assumption that it is hard to factor a large number or to compute a discrete logarithm. We do not care about those implementation details, and thus abstract over them by defining a single encryption operation that encrypts an arbitrary term with a key and can only be decrypted if the inverse of the key is known to the recipient. We ignore all types of attacks that are based on mathematical implementation details or use a side-channel to extract information about the plaintext, encryption is treated as a perfect black box.

For that to work we first define the inverse of a term.

Definition 4.4 (Inverse term)**.** *For $x, y \in \mathsf{Agent} \cup \mathsf{Role}$ we define the inverse relation:*

$$\begin{aligned} \mathrm{pk}(x)^{-1} &= \mathrm{sk}(x) \\ \mathrm{sk}(x)^{-1} &= \mathrm{pk}(x) \\ \mathrm{k}(x,y)^{-1} &= \mathrm{k}(x,y) \end{aligned}$$

Encrypted messages can only be decrypted if the inverse key is known to the recipient.

After we have learned to create terms out of parts, we define how to destructure terms back into those parts.

Definition 4.5 (Syntactic subterms)**.** *Let $t \in \mathsf{Term}$, then we write $t' \sqsubseteq t$ if and only if $t' \in \mathsf{Term}$ and t' is a syntactic subterm of t according the recursive specification in Definition 4.3.*

A syntactic subterm that is also a variable is called a free variable.

Definition 4.6 (Free variables)**.** *Define the free variables of a term $t \in \mathsf{Term}$ as*

$$\mathrm{FV}(t) \coloneqq \left\{ t' \;\middle|\; t' \sqsubseteq t \wedge t' \in \mathsf{Var} \cup \{v\#\mathrm{tid} \mid v \in \mathsf{Var} \wedge \mathrm{tid} \in \mathsf{TID}\} \right\}$$

Example 4.1 (Syntactic subterms and free variables)**.** *For the term $t = \{(x, Y)\}_k$, with $x \in \mathsf{Fresh}$, k a key and $Y \in \mathsf{Var}$, we have $t \sqsubseteq t$, $(x, Y) \sqsubseteq t$, $x \sqsubseteq t$, $Y \sqsubseteq t$ and $k \sqsubseteq t$. The free variables of this term are $\mathrm{FV}(t) = \{Y\}$.*

Now we will formalize the notion of a known term with the inference relation $\vdash$, which includes decryption and destructuring.

Definition 4.7 (Term Inference)**.** *Define the inference relation $\vdash$, with $M \vdash t$ for $M \subseteq \mathsf{Term}$ and $t \in \mathsf{Term}$, as the smallest relation which satisfies the recursive definition:*

$$\begin{aligned} t \in M &\implies M \vdash t && \text{base case} \\ \left(\forall t' \sqsubseteq t : t' \neq t \wedge M \vdash t'\right) &\implies M \vdash t && \text{can always construct new terms} \\ M \vdash (t_1, t_2) &\implies M \vdash t_1 \wedge M \vdash t_2 && \text{can freely unpack pairs} \\ M \vdash \{t_1\}_{t_2} \wedge M \vdash t_2^{-1} &\implies M \vdash t_1 && \text{can only decrypt if inverse key is known} \end{aligned}$$

Note how it is always possible to combine known terms into new terms, but unpacking terms is only possible when certain preconditions are met. Additionally, function calls can never be unpacked.

Now let us define security protocols and their constituents.

Definition 4.8 (Substitution)**.** *We introduce substitutions σ as functions that recursively replace all occurrences of a specified term in a given term, action, event or sequences thereof with another term. A single substitution can be written as* $[\text{new}/\text{old}]$*, where* $\text{old} \in \mathsf{BasicTerm}$ *and* $\text{new} \in \mathsf{Term}$.

Disjunct substitutions can be combined into a new substitution with the set union operator $\cup$. Each substitution has a domain $\text{dom}(\sigma) = \text{old}$ *and a range* $\text{ran}(\sigma) = \text{new}$*. The set of all substitutions is called $\mathscr{S}$.*

Example 4.2 (Substitution)**.** *Let $x, y \in \mathsf{Var}$, $r, s \in \mathsf{Role}$, $f, f_1, f_2 \in \mathsf{Fresh}$ and $\sigma = [f, s, f_2/x, r, f_1]$ be a substitution. When applied to some terms this yields:*

$$\begin{aligned}
\sigma(x) &= f \\
\sigma(f) &= f \\
\sigma(r) &= s \\
\sigma((f_1, f_2)) &= (f_2, f_2) \\
\sigma\big(\big(x, \{(f_1, f_2)\}_{\text{pk}(r)}\big)\big) &= \big(f, \{(f_2, f_2)\}_{\text{pk}(s)}\big)
\end{aligned}$$

Definition 4.9 (Actions and Events)**.** *We define the following actions and events:*

$$\begin{aligned}
\mathsf{Action} \coloneqq\ & \{\mathsf{send}(t) \mid t \in \mathsf{Term}\} \\
& \cup \{\mathsf{recv}(t) \mid t \in \mathsf{Term}\} \\
\mathsf{Event} \coloneqq\ & \{\mathsf{create}(r, \sigma) \mid r \in \mathsf{Role} \wedge \sigma \in \mathscr{S}\} \\
& \cup \mathsf{Action}
\end{aligned}$$

Actions are used in the definition of protocols, they describe what an agent may do, while events are used in the definition of the protocol's execution semantic.

And finally we can define what a security protocol is.

Definition 4.10 (Protocol)**.** *A protocol P is a partial function from the set of roles to the set of action sequences. Let $R \subseteq \mathsf{Role}$ and let $[\mathsf{Action}]$ be the set of all action sequences, then*

$$P : R \to [\mathsf{Action}]$$

The protocol definition must not contain values taken from $\mathsf{AdvConst}$.

Example 4.3 (Simple protocol)**.** *Let* $\{\text{Init}, \text{Recp}\} \subseteq \mathsf{Role}$, $\text{key} \in \mathsf{Fresh}$ *and* $x \in \mathsf{Var}$*. Then we can define a simple protocol* P*:*

$$P(\text{Init}) = \left[\mathsf{send}\big(\text{Init}, \text{Recp}, \{\{\text{Recp}, \text{key}\}_{\text{sk}(\text{Init})}\}_{\text{pk}(\text{Recp})}\big)\right]$$
$$P(\text{Recp}) = \left[\mathsf{recv}\big(\text{Init}, \text{Recp}, \{\{\text{Recp}, x\}_{\text{sk}(\text{Init})}\}_{\text{pk}(\text{Recp})}\big)\right]$$

Its sequence chart can be seen in Figure 1.

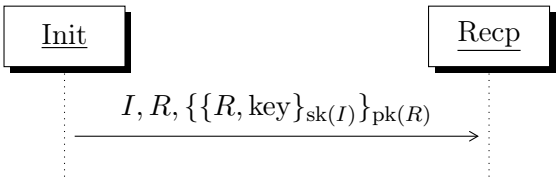

Figure 1: Sequence diagram of Example 4.3, key is a freshly generated nonce

4.2 Protocol execution

Localization and threads are used to define the execution rules of security protocols.

Definition 4.11 (Localization function)**.** *Define the localization substitution* $\text{localize} : \mathsf{TID} \to \mathscr{S}$ *for a thread identifier* $\text{tid} \in \mathsf{TID}$ *as:*

$$\text{localize}(\text{tid}) := \bigcup_{v \in \mathsf{Fresh} \cup \mathsf{Var}} [v\#\text{tid}/v]$$

Localization adds a suffix to each identifier to mark its origin.

Definition 4.12 (Thread creation function)**.** *Let* $l \in [\mathsf{Action}]$ *be an action sequence,* $\text{tid} \in \mathsf{TID}$ *be the thread identifier and* $\sigma \in \mathscr{S}$ *be the role-to-agent mapping. Define the thread creation function* $\text{thread} : ([\mathsf{Action}] \times \mathsf{TID} \times \mathscr{S}) \to [\mathsf{Action}]$ *as:*

$$\text{thread}(l, \text{tid}, \sigma) := \sigma(\text{localize}(\text{tid})(l))$$

Given the action sequence from the protocol definition, a new thread identifier and the desired role-to-agent mapping, this function will translate the action sequence in preparation for execution by localizing all identifiers and replacing role names with concrete agents.

Example 4.4 (Thread creation)**.** *Let* P *be the protocol from Example 4.3,* $\text{tid} \in \mathsf{TID}$ *and* $\{A, B\} \subseteq \mathsf{Agent}$*. When creating a thread which executes the* Init *role as* A *and the* Recp *role as* B*, we have*

$$\text{localize}(\text{tid})(\text{key}) = \text{key}\#\text{tid}$$
$$\text{localize}(\text{tid})(x) = x\#\text{tid}$$
$$\text{thread}(P(\text{Init}), \text{tid}, [A, B/\text{Init}, \text{Recp}]) = \left[\mathsf{send}\big(A, B, \{\{B, \text{key}\#\text{tid}\}_{\text{sk}(A)}\}_{\text{pk}(B)}\big)\right]$$
$$\text{thread}(P(\text{Recp}), \text{tid}, [A, B/\text{Init}, \text{Recp}]) = \left[\mathsf{recv}\big(A, B, \{\{B, x\#\text{tid}\}_{\text{sk}(A)}\}_{\text{pk}(B)}\big)\right]$$

Definition 4.13 (Adversary Knowledge)**.** *The adversary always carries a set of terms which are known to them at the current time. Initially the adversary knowledge consists of:*

$$\begin{aligned}\mathrm{IK}_0 \coloneqq\ & \mathsf{Agent} \\ & \cup \mathsf{AdvConst} \\ & \cup \bigcup_{a \in \mathsf{Agent}} \{\mathrm{pk}(a)\} \\ & \cup \bigcup_{a \in \mathsf{Compromised}} \{\mathrm{sk}(a)\} \\ & \cup \bigcup_{\substack{a \in \mathsf{Compromised} \\ b \in \mathsf{Agent}}} \{\mathrm{k}(a,b), \mathrm{k}(b,a)\}\end{aligned}$$

Where $\mathsf{Compromised} \subseteq \mathsf{Agent}$ *is the set of compromised agents.*

The adversary initially knows all adversary generated constants, all agent names, their respective public keys and all keys of any compromised agent.

Definition 4.14 (Execution)**.** *Now we can finally define the state-based execution rules. Let* $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ *be the state with* $\mathrm{tr} \in [\mathsf{TID} \times \mathsf{Event}]$ *the current trace,* $\mathrm{IK} \subseteq \mathsf{Term}$ *the current adversary knowledge and* $\mathrm{th} \in \mathsf{TID} \to [\mathsf{Event}]$ *the remaining events in a thread. The initial state is defined as* $s_0 \coloneqq ([], \mathrm{IK}_0, \emptyset)$*.*

The transitions are defined as follows, with preconditions stated above the line and actions below it. The $\oplus$ *operator means sequence concatenation.*

$\mathsf{create}(R, \sigma)$*:*

$$\frac{R \in \mathrm{dom}(P) \qquad \sigma \in \mathrm{dom}(P) \to \mathsf{Agent} \qquad \mathrm{tid} \notin \mathrm{dom}(\mathrm{th})}{(\mathrm{tr}, \mathrm{IK}, \mathrm{th}) \mapsto (\mathrm{tr} \oplus [(\mathrm{tid}, \mathsf{create}(R, \sigma))], \mathrm{IK}, \mathrm{th}[\mathrm{tid} \mapsto \mathrm{thread}(P(R), \mathrm{tid}, \sigma)])}$$

This transition will start a thread of the protocol P *from the view of role* R *with the agent-to-role assignment given by* σ*. The events to be executed are added to the state variable* th*. Note that there are no constraints on the exact nature of the role-to-agent assignment* σ*. A thread does not need to have an opposite, which would use the same substitutions from the view of a different role. The same thread can be started multiple times with a different thread identifier. Additionally, infinite threads can be created.*

$\mathsf{send}(m)$*:*

$$\frac{\mathrm{th}(\mathrm{tid}) = [\mathsf{send}(m)] \oplus l}{(\mathrm{tr}, \mathrm{IK}, \mathrm{th}) \mapsto (\mathrm{tr} \oplus [(\mathrm{tid}, \mathsf{send}(m))], \mathrm{IK} \cup \{m\}, \mathrm{th}[\mathrm{tid} \mapsto l])}$$

The send *event transmits a message by adding it to the adversary knowledge. The Dolev-Yao adversary fully controls the network and acts as the messenger.*

$\mathsf{recv}(\mathrm{pt})$*:*

$$\frac{\mathrm{th}(\mathrm{tid}) = [\mathsf{recv}(\mathrm{pt})] \oplus l \qquad \mathrm{IK} \vdash \sigma(\mathrm{pt}) \qquad \mathrm{dom}(\sigma) = \mathrm{FV}(\mathrm{pt})}{(\mathrm{tr}, \mathrm{IK}, \mathrm{th}) \mapsto (\mathrm{tr} \oplus [(\mathrm{tid}, \mathsf{recv}(\mathrm{pt}))], \mathrm{IK}, \mathrm{th}[\mathrm{tid} \mapsto \sigma(l)])}$$

The recv *event receives a message by deriving it from the current adversary knowledge. Thus, the adversary can control what messages arrive and may change them. The substitution* σ *represents the variable assignment that makes the received message term match the pattern* pt *which may contain free variables. Those variable assignments are distributed down the chain by applying the substitution to the remaining events. Note that the received message can be an arbitrary derived term that was not sent by another thread. If it is not possible to derive a message that fits the pattern* pt*, this action cannot be executed which might lead to a deadlock.*

Note that all threads execute at the same time, they are interleaved*.*

Example 4.5 (Protocol execution)**.** *Let* P *be the protocol from Example 4.3,* $t_1 \in \mathsf{TID}$ *and* $\{A, B\} \subseteq \mathsf{Agent}$*. An execution of this protocol using the rules from Definition 4.14 could look like this:*

1. *The initial state is* $s_0 = ([], \{A, B, \mathrm{pk}(A), \mathrm{pk}(B)\}, \{\})$*.*
2. *The first transition is* `create`*, with* $R = \mathrm{Init}$*,* $\mathrm{tid} = t_1$ *and* $\sigma = \sigma_1 = [A, B/\mathrm{Init}, \mathrm{Recp}]$*. Then the next state is:*

$$s_1 = \Big([(t_1, \mathsf{create}(\mathrm{Init}, \sigma_1))], \; \{A, B, \mathrm{pk}(A), \mathrm{pk}(B)\}, \; \{t_1 \mapsto [\mathsf{send}(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)})]\} \Big)$$

3. *To create the corresponding recipient thread we use* `create` *with* $R = \mathrm{Recp}$*,* $\mathrm{tid} = t_2$ *and* $\sigma = \sigma_1$*. Then the next state is:*

$$s_2 = \Big([(t_1, \mathsf{create}(\mathrm{Init}, \sigma_1)), (t_2, \mathsf{create}(\mathrm{Recp}, \sigma_1))], \; \{A, B, \mathrm{pk}(A), \mathrm{pk}(B)\}, \; \{t_1 \mapsto [\mathsf{send}(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)})], \; t_2 \mapsto [\mathsf{recv}(A, B, \{\{B, x\#t_2\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)})]\} \Big)$$

4. *After both threads have been created the actual execution can begin. The thread* t_1 *will start with* `send`*, thus* $\mathrm{tid} = t_1$ *and* $m = (A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)})$*. The next*

state is:

$$s_3 = \Big(\big[(t_1, \mathsf{create}(\mathrm{Init}, \sigma_1)), (t_2, \mathsf{create}(\mathrm{Recp}, \sigma_1)), \\ \big(t_1, \mathsf{send}\big(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big) \big], \\ \big\{A, B, \mathrm{pk}(A), \mathrm{pk}(B), \big(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big\}, \\ \big\{t_1 \mapsto [], t_2 \mapsto \big[\mathsf{recv}\big(A, B, \{\{B, x\#t_2\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big]\big\} \Big)$$

5. *And finally the message will be received by the thread t_2. Let* $\mathrm{tid} = t_2$ *and* $\mathrm{pt} = \big(A, B, \{\{B, x\#t_2\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)$. *For the precondition* $\mathrm{IK} \vdash \sigma(\mathrm{pt})$ *to hold, we set* $\sigma = [\mathrm{key}/x]$. *The final state is:*

$$s_4 = \Big(\big[(t_1, \mathsf{create}(\mathrm{Init}, \sigma_1)), (t_2, \mathsf{create}(\mathrm{Recp}, \sigma_1)), \\ \big(t_1, \mathsf{send}\big(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big), \\ \big(t_2, \mathsf{recv}\big(A, B, \{\{B, \mathrm{key}\#t_2\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big) \big], \\ \big\{A, B, \mathrm{pk}(A), \mathrm{pk}(B), \big(A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}\big)\big\}, \\ \{t_1 \mapsto [], t_2 \mapsto []\} \Big)$$

The execution is finished now, because the action lists for each role are empty.

4.3 Protocol properties

Using these rules we create a transition system for a given protocol P. This model can be implemented via the B method for animation and model checking. But first some properties are required that a protocol needs to fulfill. We formulate these properties based on the current state $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$, and a protocol fulfills a property if and only if all states reachable from the initial state s_0, using the transition rules defined in Definition 4.14, fulfill the given property.

First we define an auxiliary property which is used to specify the concrete properties, namely thread honesty.

Definition 4.15 (Honest thread)**.** *We define a thread to be honest in a state s, if and only if it is finished and no compromised agents are involved. Let $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ be a state.*

$$\mathrm{HT}(s, \mathrm{tid}) \coloneqq \exists\, (R, \sigma) \in \mathsf{Role} \times \mathscr{S} : \\ (\mathrm{tid}, \mathsf{create}(R, \sigma)) \in \mathrm{tr} \wedge \mathrm{th} = [] \wedge (\mathrm{ran}(\sigma) \cap \mathsf{Compromised}) = \emptyset$$

An important property of a protocol is the secrecy of shared passwords or keys.

Definition 4.16 (Secrecy)**.** *A state* $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ *satisfies secrecy of a secret* $t \in \mathsf{Fresh}$*, if and only if*

$$\forall \mathrm{tid} \in \mathsf{TID} : \mathrm{HT}(s, \mathrm{tid}) \implies \neg\,(\mathrm{IK} \vdash t\#\mathrm{tid})$$

The adversary may not deduce the secret t*, localized to all honest threads, from its knowledge.*

Another important property is authentication, which gives an agent guarantees for the identity of their communication partner. There are multiple formulations for this property that have different strengths.

Definition 4.17 (Weak aliveness)**.** *A state* $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ *satisfies weak aliveness of a role* $R \in \mathsf{Role}$*, if and only if*

$$\begin{aligned}\forall\,(\mathrm{tid}', R', \sigma') \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : \big(\mathrm{HT}\big(s, \mathrm{tid}'\big) \wedge \big(\mathrm{tid}', \mathsf{create}\big(R', \sigma'\big)\big) \in \mathrm{tr}\big)\\ \implies \big(\exists\,(\mathrm{tid}, R, \sigma) \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : (\mathrm{tid}, \mathsf{create}(R, \sigma)) \in \mathrm{tr} \wedge \sigma'(R) = \sigma(R)\big)\end{aligned}$$

When an honest thread thinks they communicated with an agent in role R*, that agent started the protocol in role* R *before.*

The following properties are based on the authentication properties presented by Lowe [Low97].

Definition 4.18 (Weak agreement)**.** *A state* $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ *satisfies weak agreement of a role* $R \in \mathsf{Role}$*, if and only if*

$$\begin{aligned}\forall\,(\mathrm{tid}', R', \sigma') \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : \big(\mathrm{HT}\big(s, \mathrm{tid}'\big) \wedge \big(\mathrm{tid}', \mathsf{create}\big(R', \sigma'\big)\big) \in \mathrm{tr}\big)\\ \implies \big(\exists\,(\mathrm{tid}, R, \sigma) \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : (\mathrm{tid}, \mathsf{create}(R, \sigma)) \in \mathrm{tr} \wedge \sigma'(R) = \sigma(R)\\ \wedge\, \sigma'\big(R'\big) \in \mathrm{ran}(\sigma)\big)\end{aligned}$$

When an honest thread thinks they communicated with an agent in role R*, that agent started the protocol in role* R *before and they communicated with the honest agent in any role.*

Definition 4.19 (Simple agreement)**.** *A state* $s = (\mathrm{tr}, \mathrm{IK}, \mathrm{th})$ *satisfies simple agreement of a role* $R \in \mathsf{Role}$*, if and only if*

$$\begin{aligned}\forall\,(\mathrm{tid}', R', \sigma') \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : \big(\mathrm{HT}\big(s, \mathrm{tid}'\big) \wedge \big(\mathrm{tid}', \mathsf{create}\big(R', \sigma'\big)\big) \in \mathrm{tr}\big)\\ \implies \big(\exists\,(\mathrm{tid}, R, \sigma) \in \mathsf{TID} \times \mathsf{Role} \times \mathscr{S} : (\mathrm{tid}, \mathsf{create}(R, \sigma)) \in \mathrm{tr} \wedge \sigma'(R) = \sigma(R)\\ \wedge\, \sigma'\big(R'\big) = \sigma\big(R'\big)\big)\end{aligned}$$

When an honest thread thinks they communicated with an agent in role R*, that agent started the protocol in role* R *before, and they communicated with the honest agent in the correct role.*

Example 4.6 (Properties of the example protocol)**.** *The example trace given in Example 4.5, coming from the protocol specified in Example 4.3, fulfills the secrecy property for the fresh value* key*. The only state that is honest is the final state* s_4*, because the protocol is finished*

and no compromised agents are participating. The adversary knowledge in this state is $\mathrm{IK}_4 = \{A, B, \mathrm{pk}(A), \mathrm{pk}(B), (A, B, \{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)})\}$*. Secrecy of* key *would not be fulfilled if it were possible to infer* $\mathrm{key}\#t_1$ *from* IK_4*, but* $\mathrm{IK}_4 \nvdash \mathrm{key}\#t_1$*, because the secret key* $\mathrm{sk}(B)$ *that is needed to decrypt the message term* $\{\{B, \mathrm{key}\#t_1\}_{\mathrm{sk}(A)}\}_{\mathrm{pk}(B)}$ *is not known and not inferable for the adversary.*

But this alone is not sufficient to prove the secrecy of key*. Not only a single trace, but all valid traces and the states within must exhibit the secrecy property. In this simple case, because there is only ever one message and this message is always asymmetrically encrypted with* $\mathrm{pk}(B)$*, which needs the uninferable* $\mathrm{sk}(B)$ *to decrypt, this simple protocol* does *fulfill the secrecy property for* key*.*

5 A DSL for security protocols

A DSL (domain-specific language) is a language that is specialized for a singular domain. In this case modeling security protocols and their properties for translation into a form that can be used by model checking tools

This Dolev-Yao DSL is built from EDN data structures, so we inherit the simple parsing inherent to Clojure code itself. It mirrors the DSL implemented by *lisb* to define B code. After some checks and transformations which are explained in Section 6, a machine is generated using the specified formalism.

The Dolev-Yao DSL's syntax and semantic closely resemble the definition of the Dolev-Yao formal model given in Section 4. All Clojure symbols reference built-in Dolev-Yao features, while Clojure keywords represent user-defined names.

5.1 Structure

A protocol definition consists of a name, `threads` (see Section 5.2) and `properties` (see Section 5.4). They are arranged as shown in Listing 6.

Listing 6: Dolev-Yao DSL structure

```
1: (protocol
2:  <NAME> ;; Protocol name as a keyword
3:  (threads <THREAD_1> ... <THREAD_N>) ;; List of threads
4:  (properties <PROPERTY_1> ... <PROPERTY_N>)) ;; Properties to check
```

5.2 Threads

The `threads` structure corresponds to Definition 4.10, it defines participants of the protocol and their actions. A thread definition is a list of two elements, the name of the acting role and a vector of events to execute in order. The syntax is shown in Listing 7.

Listing 7: Thread structure

```
1: (<ROLE> [<EVENT_1> ... <EVENT_N>])
```

5.2.1 Actions

The Dolev-Yao DSL defines three different actions: `generate`, `send` and `receive`.

`generate` is a new action that has no counterpart in Section 4. It exists to simplify the handling of free variables (from Var) and fresh values (from Fresh). Both of these sets are defined in Definition 4.1. In the formal model a identifier in a term can refer to a free variable, a fresh value or a constant. Mathematically they can be differentiated by checking their set membership with Var, Fresh or Func. For simplicity and ease of use, all keywords in the Dolev-Yao DSL refer to variables. When sending a term it must be fully ground, meaning all of those variables must have a value. A value for a unknown variable is not automatically generated, so the `generate` event will fill it with a guaranteed unique value, which serves the same purpose as declaring it an element of Fresh. Currently, there is no direct support for constants, although it is possible to use variables for the same purpose.

The `send` and `receive` actions correspond to the actions of the same name defined in Definition 4.9 with the semantic given in Definition 4.14.

Listing 8: Action structure

```
1: (generate <VARIABLE>)
2: (send <TERM>)
3: (receive <TERM>)
```

5.3 Terms

The `send` and `receive` actions require a term as a message pattern. They are defined in Definitions 4.2 and 4.3 and this Dolev-Yao DSL follows that closely, with all implemented terms listed in Table 1. Terms can be arbitrarily nested. There is no analog to sending an agent or role as a term, this is not needed as the public key of an agent is static and known to all agents and can thus be used as a substitute for the identity.

Dolev-Yao DSL term	Purpose
`(sequence T_1 ... T_N)`	Contains an ordered list of terms.
`(enc Key Term)`	Represents encryption of a term with a given key.
`(hash Term)`	The application of a one-way hash function.
`(public-key :role)`	The public key of the given role or actor, used for encryption.
`(secret-key :role)`	The secret key of the given role or actor, used for encryption.
`(shared-key :role1 :role2)`	The shared key between the given roles or actors, used for encryption.
`:name` or `(var :name)`	A logical variable, may be given a value with the `generate` or `receive` actions.
Fresh values	Used at runtime to represent the unique term created from `generate`.

Table 1: Dolev-Yao DSL terms and their purpose

5.4 Properties

The Dolev-Yao DSL implements the secrecy and authentication properties listed in Definitions 4.16 to 4.19.

Listing 9: Property structure

```
1: (secrecy <SCOPED_VARIABLE_1> ... <SCOPED_VARIABLE_N>)
2: (weak-aliveness <ROLE_1> ... <ROLE_N>)
3: (weak-agreement <ROLE_1> ... <ROLE_N)
4: (simple-agreement <ROLE_1> ... <ROLE_N)
```

As can be seen in Listing 9, the property definitions all have a similar structure. They consist of their type followed by their arguments in a list.

The secrecy property takes the variables whose values are supposed to stay unknown to the adversary. Those variable names are given as Clojure keywords, namespaced to the role in which they are defined in. For instance the keyword `:x` refers to a local variable called x in the current role, while `:Init/x` refers to the variable x declared in the role `Init`.

The authentication properties take the role mentioned in their respective definitions as arguments.

5.5 Semantic of execution

The semantic closely follows the Dolev-Yao formal model and the transition rules given in Definition 4.14. The execution is split into two parts, first the participating threads are created with a role-to-agent mapping and then each thread follows the actions of their respective role. As described in the transition rules, the creation of one thread is modeled as one operation. Then another operation ends the thread creation phase and switches the protocol into the execution phase. In this phase each operation corresponds to one action defined in the protocol, like `send`, `receive` or `generate`. The different backends might add additional operations if required.

5.6 Example

Listing 10 contains the translation of the protocol given in Example 4.3 to the Dolev-Yao DSL.

Note the `(generate :key)` in line 4. All variables inside sent terms must have a value, either from a previous `receive` action or from a `generate` action. This removes ambiguities and simplifies the implementation.

But term patterns that are passed to `receive` actions may write to new variables, as seen in this example, or compare the received terms with the value of already written variables. This mirrors Prolog's term unification semantic.

Listing 10: Simple protocol translated from Example 4.3

```
1:  (protocol
2:   :simple_example
3:   (threads
4:    (:Init [(generate :key)
5:            (send (sequence (public-key :Init)
6:                            (public-key :Recp)
7:                            (enc (secret-key :Init)
8:                                 (enc (public-key :Recp)
9:                                      (sequence (public-key :Recp)
10:                                               :key))))))])
11:   (:Recp [(receive (sequence (public-key :Init)
12:                              (public-key :Recp)
13:                              (enc (secret-key :Init)
14:                                   (enc (public-key :Recp)
15:                                        (sequence (public-key :Recp)
16:                                                  :x))))))]))
17:  (properties
18:   (secrecy :Init/key)))
```

6 Translation

To translate the given Dolev-Yao DSL we develop a new tool called *dy2prob*, which is a command-line application written in Clojure that parses a protocol description, applies some checks and transformations and outputs a translation based on the selected mode and options. An overview of the process is shown in Figure 2.

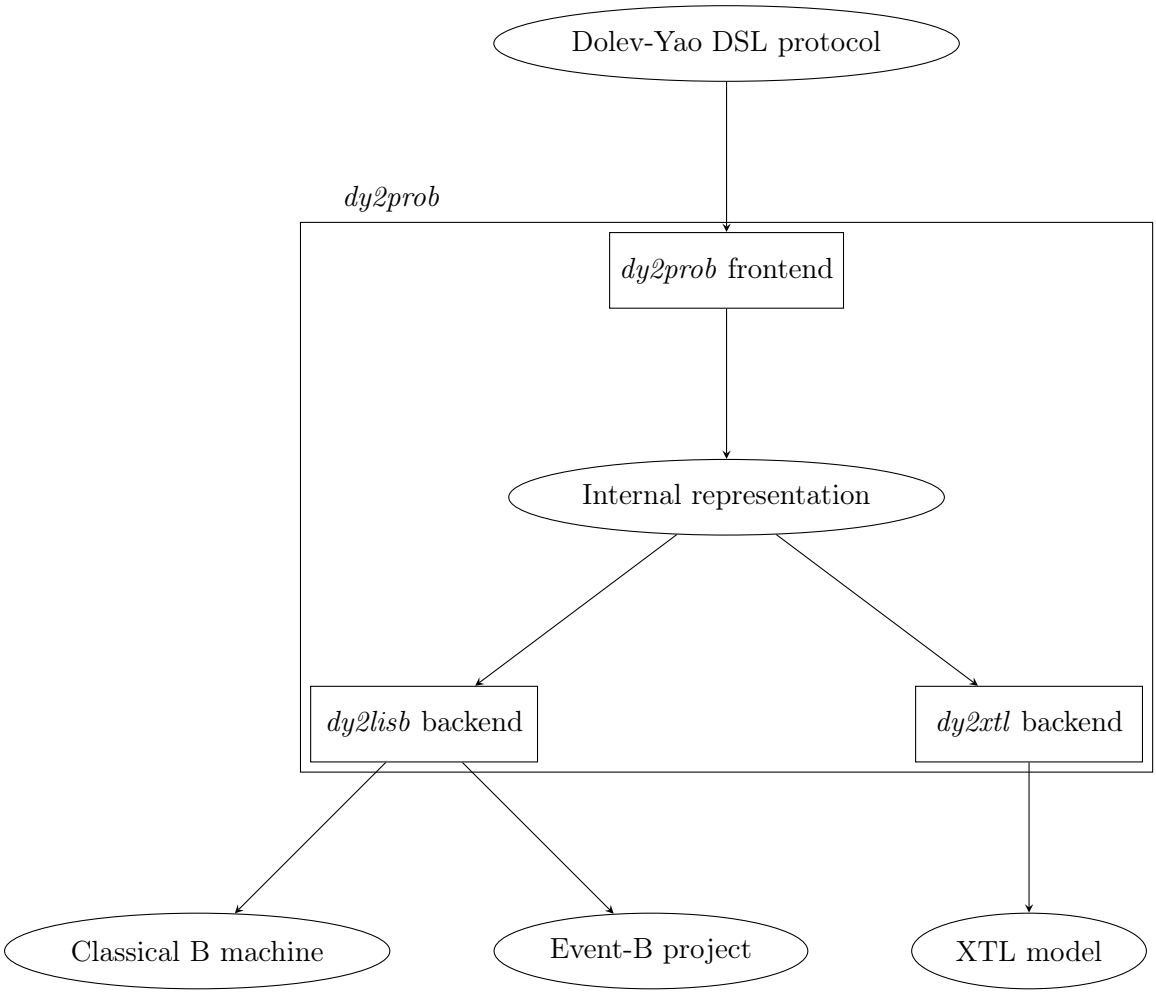

Figure 2: *dy2prob* translation process

After parsing the Dolev-Yao DSL, the user-facing representation is translated into a simpler and more verbose intermediate representation. Then the following transformations are run.

First all used variables are identified, collected and saved into a set for later use. Each variable name is prefixed with its owner, the role which runs the thread this variable term is part of, for scoping purposes. This is also the scope used in the definition of the secrecy

property, it must be prefixed as a namespace to the Clojure keyword.

Then all variable uses are annotated with metadata, namely `:var-info` which may contain `:read` for reading variable access and `:write` for writing variable access.

During this iteration all variable uses are checked for correctness. Reading is only possible after the variable has been written to and redefining variables is not possible. So `generate` requires an unwritten variable, `send` a written variable and `receive` can deal with both.

Dolev-Yao DSL term	Internal representation
`(sequence T1 T2 T3)`	`{:type :sequence, :terms [T1 T2 T3]}`
`(enc Key Term)`	`{:type :enc, :key Key, :term Term}`
`(hash Term)`	`{:type :hash, :term Term}`
`(public-key :role)`	`{:type :public-key, :role :role}`
`(secret-key :role)`	`{:type :secret-key, :role :role}`
`(shared-key :role1 :role2)`	`{:type :shared-key, :roles [:role1 :role2]}`
`:name` or `(var :name)`	`{:type :var, :variable :name}`
Fresh values	Only used at runtime, so there is no internal representation.

Table 2: Internal Clojure representations of Dolev-Yao terms

The generated intermediate representation is then passed to one of two different backends depending on user selection. One that outputs Prolog code for an XTL model and one that outputs a *lisb* representation of a B model.

6.1 Common backend properties

All backends will generate models that can be animated and model checked. But when comparing them to the formal mathematical definition given in Definition 4.14 there are some differences.

The first one is the split into two phases, in the beginning the creation of all threads and after that the execution of the protocol steps, while the formal model allows the creation of a new thread at any time. This improves the user's comprehension of the current state of the model, while having no significant impact on the performance of model checking. Exactly how this change impacts model checking can be seen in Section 9.1. The thread creation also does not allow arbitrary role-to-agent mappings but instead uses manually implemented symmetry reduction to reduce the number of equivalent setups, for more information refer to Section 9.2.

Another difference is the limitation of the thread count to a finite number, whereas in the formal model there is no limit to the number of threads that can exist at the same time.

The purpose of this is to allow explicit state model checking on a finite state-space, as explained in Section 9.3.

Finally, variables are only allowed to contain fresh values like nonces. This massively simplifies the implementation of pattern matching when receiving a term, else there would be infinitely many possible terms that could be received for any variable in a `receive` message pattern. No protocol that was modeled by this Dolev-Yao DSL required variables to contain more complicated terms. Additionally, the concept of AdvConst was removed, as it is superfluous when the adversary can just generate a new and unused Fresh value when required.

When executing a security protocol, the Dolev-Yao semantic demands substitutions every time a message is received to replace variables with another term. Due to limitations of the classical B language, it is not possible to efficiently implement such a recursive replacement. Thus, each thread receives a memory field, which holds the values of each variable known to them. When constructing a term, the memory is queried for the value of all variables and those values are used instead of the variables. On the other hand, when receiving a message or generating a fresh value, that value is either compared with the existing value or saved into memory.

The Dolev-Yao semantic is defined using the *localize* function, which appends the thread's identifier as a suffix to each fresh value or variable, as a way to track them across substitutions. Because the substitution approach is not used in favor of thread-local memory, a localization is not necessary. Variables are identified by the role that declares them, and each fresh value generated is unique and saved in the generating thread's memory. Thus, they can be tracked by the security properties. Especially in the *secrecy* property, which uses localized variables to identify the secrets in the formal model, but now tracks them by value.

Some features of the ProB toolchain work better with different representations. Visualization and animation should use verbose representations and model checking requires performance. Translating the model into different formalisms, like Event-B or TLA^+, imposes restrictions because of unsupported syntax constructs or features. Thus, multiple translations are required, which can be controlled by configuration options. By selecting a different output language and a preset, one can control what the output of the generation process will look like. The supported modes and presets are listed in Table 3 and explained further in the following sections.

Backend	Mode	Preset	Description
dy2lisb	`b`	`b` (default)	B machine optimized for compatibility
		`visualization`	B machine built for better user comprehension during animation and visualization
		`tla`	B machine optimized for further translation to TLA$^+$
	`eventb`	`eventb` (default)	Event-B model
dy2xtl	`xtl`	`xtl` (default)	XTL model

Table 3: Output modes and presets

The *dy2prob* application can be called from the command-line as shown in Listing 11. The mode is selected with the `-m` switch and the preset is selected with the `-p` switch. Additional options can be passed with the `-O` switch. The output directory can be changed with the `-o` switch. Then the input files or directories containing the protocols are listed.

Listing 11: Command-line help of *dy2prob*

```
Dolev-Yao -> ProB.

Usage: dy2prob [options] <INPUT FILE(S) OR DIRECTORY>

Options:
-h, --help                  Show help
-o, --output DIR      .     Output directory,
                            defaults to current directory
-m, --mode MODE             Output mode, one of: b, eventb, xtl
-p, --preset PRESET   nil   Preset options for translation,
                            will default to output mode,
                            one of b, eventb, tla, visualization, xtl
-O, --option K[=V]    {}    Extra options for translation
```

All backends support the `:max-threads` and `:max-agents` properties, which control how many threads are allowed to be created and from how many agents the role-to-agent assignment can choose, which is relevant for the size of the state-space as explained in Sections 9.2 and 9.3

6.2 XTL-Backend (*dy2xtl*)

6.2.1 XTL format

The XTL format[3] is a way to write down state-based models as Prolog programs that can be loaded and processed by the ProB toolchain. The models must be valid SICStus Prolog programs, and they must implement the `start/1`, `trans/3` and `prop/2` predicates, where `<name>/N` refers to a defined predicate with the given name that takes N parameters.

`start(-InitialState)` defines the initial states. A state can be an arbitrary ground term.

`trans(?Transition, +OldState, -NewState)` lists all possible transitions and the destination state given the current state. A transition can be an arbitrary ground term.

`prop(+State, -Property)` is used to query the properties of a given state. A property is an arbitrary ground term. This can be used to encode state-based invariants with the special `unsafe` property. Model checking will find states that have the `unsafe` property set and mark them as invariant violations.

A simple model of a counter is shown in Listing 12, which is equivalent to Listing 1. Note that transitions can also be compound terms with arguments. When using logical variables in a `trans/3` clause inside the transition term, all possibles assignments of these variables should be obtainable when using backtracking.

Listing 12: XTL specification modeling a counter

```
1:  :- use_module(library(between)).
2:
3:  start(0).
4:
5:  trans(inc, N, Next) :-
6:  Next is N+1.
7:  trans(inc_by(Delta), N, Next) :-
8:  between(1, 3, Delta),
9:  Next is N+Delta.
10: trans(dec, N, Next) :-
11: N > 0,
12: Next is N-1.
13:
14: prop(N, unsafe) :-
15: N < 0.
```

[3] https://prob.hhu.de/w/index.php?title=Other_languages

6.2.2 Generating Prolog code from Clojure

The goal is to create a valid Prolog program from Clojure code. Working with strings directly would be highly inadvisable because of implementation and maintenance difficulties. Additionally, the built-in string processing facilities of the Clojure standard libarary are poor and one must use the Java host functions. But the probparsers project contains a Prolog library written in Java, which allows easy creation of Prolog terms, and due to Clojure's strengths it is trivial to develop a new DSL for Prolog terms. Prolog itself also follows the *code is data, data is code* approach and thus Prolog code just consists of Prolog terms. Although there is a difference between the readable notation that a programmer would write and the canonical representation, which does not allow infix operators and operator precedence.

Listing 13: Clojure representation of the counter XTL model in Listing 12

```
1:  [(directive '(:use_module (:library :between)))
2:   (clause '(:start 0))
3:   (clause '(:trans :inc N Next)
4:           '(:is Next (:+ N 1)))
5:   (clause '(:trans (:inc_by Delta) N Next)
6:           '(:between 1 3 Delta)
7:           '(:is Next (:+ N Delta)))
8:   (clause '(:trans :dec N Next)
9:           '(:> N 0)
10:          '(:is Next (:- N 1)))
11:  (clause '(:prop N :unsafe)
12:          '(:< N 0))]
```

An example of the developed DSL can be seen in Listing 13, which generates a Prolog program equivalent to Listing 12. Clojure's keywords correspond to Prolog atoms, while numbers and strings are translated to Prolog numbers and strings respectively. Identifiers in Clojure are called symbols, and they are translated into Prolog variables, but they must start with an uppercase letter or an underscore, as Prolog does not allow variables in quotes. Clojure's vectors correspond to Prolog lists, and the `|` symbol is used in vectors to represent Prolog's head-tail notation of lists. A Clojure list, denoted by parentheses, represents a Prolog compound term, whose functor is the first item in the Clojure list, which must be a keyword. Each Clojure form inside a vector or list is recursively translated with the same routine. Some examples can be seen in Table 4. After fully printing a Prolog term, a dot is added to convert it into a sentence, the building blocks of a Prolog program. The calls to `directive` and `clause` are evaluated to lists with `:-` as the first element. This is necessary, because keywords containing a colon cannot be written as a literal in Clojure.

Element	Clojure DSL syntax	Prolog translation
Keyword	`:foo`	`foo`
	`:Foo`	`'Foo'`
Symbol	`Foo`	`Foo`
	`_foo`	`_foo`
	`foo`	Error
Number	`42`	`42`
	`1.5`	`1.5`
Vector	`[]`	`[]`
	`[:a]`	`[a]`
	`[:a 1 X]`	`[a,1,X]`
	`[:a 1 | X]`	`[a,1|X]`
List	`(:foo)`	`foo`
	`(:foo :a)`	`foo(a)`
	`(:foo :a 1 X)`	`foo(a,1,X)`
Nested	`[(:a (:b X) 1) [2 (:c 7.5)]|Y]`	`[a(b(X),1),[2,c(7.5)]|Y]`

Table 4: Prolog DSL translation

6.2.3 XTL representation of the Dolev-Yao model

The semantic of Prolog is very similar to the semantic of the Dolev-Yao formal model, which allows a straightforward translation using backtracking and pattern matching to generate the recursive term structures. Additionally, the semantic of Prolog's logical variables matches what happens to variables and fresh values in the Dolev-Yao world, as explained in Section 5.2.1.

The generated XTL model contains two parts, the static part which is the same for every protocol and implements the Dolev-Yao semantic with the required predicates, and the dynamic part which states the protocol and its properties in a format that is understood by the former part.

The model state in the XTL implementation is a compound term which looks like `s(Phase, Threads, AdversaryKnowledge, NextFresh)`. The contained arguments are explained in Table 5. Contrary to the state given in Definition 4.14 the trace is not saved, as each thread and its start parameters are preserved in the `Threads` list.

State argument	Description
`Phase`	Phase of protocol execution, either `create` or `run`
`Threads`	List of currently running threads
`AdversaryKnowledge`	List of terms that are known to the adversary
`NextFresh`	Counter to help generate unique terms

Table 5: Explanation of XTL `state` term arguments

`Phase` is the current protocol phase which can be either `create` or `run`. In the creation phase the threads are created with the intended role-to-agent assignments, and in the run phase each thread executes its role in the protocol.

`Threads` contains a list of the currently running threads. Each thread is a compound term of the form `thread(TID, Role, Sub, Actions, Memory)`, as explained in Table 6. `TID` contains the numeric thread identifier, `Role` is the acting role of this thread, `Sub` contains the role-to-agent mapping, `Actions` is a list of actions that still need to be executed, and `Memory` stores the values that are associated with the local variables.

Thread argument	Description
`TID`	Unique thread identifier
`Role`	Role that the thread is acting as in the protocol
`Sub`	Substitution that saves the role-to-agent mapping
`Actions`	List of actions that need to be executed
`Memory`	Thread-local memory, a list of variable names and their values

Table 6: Explanation of XTL `thread` term arguments

`AdversaryKnowledge` is a list of Dolev-Yao terms that are known to the adversary. In the beginning it is initialized to contain all public keys and the private and shared keys of all compromised agents.

`NextFresh` is a global counter used to generate unique fresh values.

The central part of the modeling is the representation of terms. Those Dolev-Yao terms are modeled by Prolog terms, as shown in Table 7.

Dolev-Yao DSL term	XTL term
`(sequence T1 T2 T3)`	`[T1, T2, T3]`
`(enc Key Term)`	`enc(Key, Term)`
`(hash Term)`	`h(Term)`
`(public-key :role)`	`pk(role)`
`(secret-key :role)`	`sk(role)`
`(shared-key :role1 :role2)`	`k(role1, role2)`
`:name` or `(var :name)`	`var(name)`
Fresh values	`fresh(N)`

Table 7: Comparison of Dolev-Yao and XTL or Prolog term representations

The dynamic part consists of the protocol definition and the protocol properties, examples of which can be seen in Listings 14 and 15 respectively. The protocol definition contains the list of actions that are executed by each role. The `send` and `receive` actions have a parameter which defines their respective message patterns. All the properties listed in Definitions 4.16 to 4.19 are implemented.

Listing 14: Protocol from Listing 10 translated into a Prolog predicate

```
%% protocol(?Role, ?ListOfActions)
protocol('Init',
         [gen(key),
          send([pk('Init'),
                pk('Recp'),
                enc(pk('Recp'),
                    enc(sk('Init'),
                        [pk('Recp'),
                         var(key)]))])]).
protocol('Recp',
         [recv([pk('Init'),
                pk('Recp'),
                enc(pk('Recp'),
                    enc(sk('Init'),
                        [pk('Recp'),
                         var(x)]))])]).
```

Listing 15: Properties from Listing 10 translated into a Prolog predicate

```
%% protocol_invariant(+State)
protocol_invariant(S) :-
    secrecy('Init', key, S). % secrecy/3 is defined in the static part
```

The maximum number of threads and agents can be dependent on the protocol and the passed options, and is therefore part of the dynamic section. All participating agents are

listed with the `honest_agent/1` and `compromised_agent/1` predicates and the maximum thread count is stated with the `max_tid/1` predicate, an example of which can be seen in Listing 16. Note that the honest agents are named `Alice` and `Bob`. Those are the standard names for agents participating in a cryptographic protocol [Sch15].

Listing 16: Generated Prolog code for the `receive` transition

```
1:  max_tid(2).
2:  honest_agent(alice).
3:  honest_agent(bob).
4:  compromised_agent(compromised1).
5:  compromised_agent(compromised2).
```

The static part defines the boilerplate required for XTL to ProB interoperability and helper predicates, which implement the transitions and properties.

The `start/1` predicate returns the initial state. The `AdversaryKnowledge` is set to the initial adversary knowledge described in Definition 4.13, and contains all public keys and the private and shared keys of all compromised agents.

The transitions defined in the `trans/3` predicate correspond to the thread actions `generate`, `send` and `receive`, but there are also transitions that control the creation of threads, namely `create`, which creates a new thread and `start`, which advances the current phase to `run`.

The `create` transition takes the role-to-agent mapping as a parameter and will create a new thread using it, while observing the maximum thread count and the agent symmetry reduction if enabled. The thread term is created, and its action list is filled with all the actions from the protocol for the specific role. The implementation can be seen in Listing 17.

Listing 17: Generated Prolog code for the `create` transition

```
1:  trans(create(TID,Role,Sub),
2:        s(create,Threads,   AdvKnowledge,NextFresh),
3:        s(create,NewThreads,AdvKnowledge,NextFresh)) :-
4:      NewThreads = [thread(TID,Role,Sub,Actions,[])|Threads],
5:      % derive TID
6:      length(Threads,ThreadCount),
7:      TID is ThreadCount+1,
8:      % check TID
9:      max_tid(MaxTID),
10:     TID =< MaxTID,
11:     % verify Role and get Actions
12:     protocol(Role,Actions),
13:     % check that Sub is a valid role-to-agent mapping
14:     check_sub(Sub).
```

The `start` transition will set the current phase to `run`, which stops the creation of new

threads but allows execution of thread actions. The implementation can be seen in Listing 18.

Listing 18: Generated Prolog code for the `start` transition

```
1: trans(start,
2:       s(create,Threads,AdvKnowledge,NextFresh),
3:       s(run,   Threads,AdvKnowledge,NextFresh)).
```

Executing the `generate`, `send` and `receive` transitions will pop the first element from the action list which must match the respective transition.

`generate` will associate the given variable with a freshly generated term using the `NextFresh` argument of the state. The variable's data is saved locally in the thread's memory, as shown in Listing 19.

Listing 19: Generated Prolog code for the `generate` transition

```
1: trans(gen(TID,Var,Term),SIn,SOut) :-
2:     pop_action(gen(Var),TID,SIn,S1),
3:     make_fresh(S1,Term),
4:     incr_next_fresh(S1,S2),
5:     set_memory(TID,Var,Term,S2,SOut).
```

`send` will take the message pattern, given as a parameter in the action term popped from the action list, replace variables using its memory and roles with its role-to-agent mapping respectively, and then add it to the adversary knowledge. All variables must be ground, so this is a deterministic operation. The implementation is presented in Listing 20.

Listing 20: Generated Prolog code for the `send` transition

```
1: trans(send(TID,Term),SIn,SOut) :-
2:     pop_action(send(Pattern),TID,SIn,S1),
3:     % replace term variables with Prolog variables
4:     % and replace roles with agents
5:     build_term_from_pattern(S1,TID,Pattern,Term,[],Vars),
6:     % get all variable values from memory
7:     get_memory_multi(TID,Vars,S1),
8:     ground(Vars), % check that all variables are set
9:     add_to_adv_knowledge(Term,S1,SOut).
```

`receive` also takes the message pattern, and does the same transformations as the `send` transition, but this time using the adversary knowledge to build the term. It must be derivable using the relation specified in Definition 4.7. Variables will be replaced by their values if the current thread has them saved in its memory, but if a variable has no value it will be assigned a fresh value from the received message. This fresh value is either newly generated or taken from the adversary knowledge, and this nondeterminism leads to

multiple possible received messages for this transition. Trying to derive terms that match a given message pattern is another part of the modeling which may lead to infinite branching. Thus variables are constrained to only ever contain fresh values to mitigate this. The Prolog code is shown in Listing 21.

Listing 21: Generated Prolog code for the `receive` transition

```
1:  trans(recv(TID,Term),SIn,SOut) :-
2:      pop_action(recv(Pattern),TID,SIn,S1),
3:      % replace term variables with Prolog variables
4:      % and replace roles with agents
5:      build_term_from_pattern(S1,TID,Pattern,Term,[],Vars),
6:      % find all valid variable assignments that the adversary can derive
7:      adversary_can_build(Term,S1,S2),
8:      ground(Vars), % check that all variables are set
9:      % unify (save and compare) with existing memory
10:     set_memory_multi(TID,Vars,S2,SOut).
```

Additionally, there are helper predicates that implement all defined security properties and get called by the `protocol_invariant` predicate based on the properties defined in the security protocol. The implementations closely follow the Dolev-Yao definitions, adjusted to fit Prolog. For example the `secrecy` predicate in Listing 22 follows Definition 4.16 and `simple_agreement` in Listing 23 resembles Definition 4.19.

Listing 22: Generated Prolog code for the `secrecy` predicate

```
1:  secrecy(Role,Var,S) :-
2:      S = s(_,Threads,AdvKnowledge,_),
3:      % for all honest threads in the role owning the variable,
4:      forall(honest_thread(thread(TID,Role,_,_,_),Threads),
5:           % the value of that variable
6:          (get_memory(TID,Var,Value,S),
7:           % is not derivable by the adversary
8:           \+ adversary_can_derive(AdvKnowledge,Value))).
```

`prop/2` defines state properties for a given XTL state. They are used to implement invariants with the `unsafe` predicate, as shown in Listing 24.

State properties are also displayed in the animator and help the user understand the current XTL state. Thus, all running threads are listed as a state property, and this can be seen in Figure 3.

Listing 23: Generated Prolog code for the `simple_agreement` predicate

```
simple_agreement(Role,S) :-
    S = s(_,Threads,_,_),
    % for all honest threads
    forall((honest_thread(thread(_,HRole,HSub,_,_),Threads),
            % which have Agent taking the given role,
            member(Role-Agent,HSub)),
            % there exists a thread started by the same agent,
           (member(thread(_,Role,Sub,_),Threads),
            member(Role-Agent,Sub),
            % and the agent that started the honest thread
            member(HRole-HAgent,HSub),
            % takes the same role in the found thread
            member(HRole-HAgent,Sub))).
```

Listing 24: Generated Prolog code for the `unsafe` clause of the `prop` predicate

```
prop(S,Safety) :-
    protocol_invariant(S) -> Safety=safe ; Safety=unsafe.
```

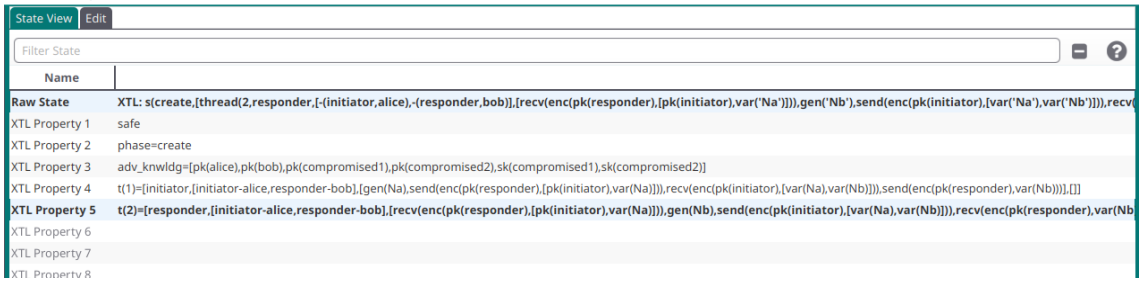

Figure 3: XTL properties visible in the ProB2-UI animator

When animating the XTL model a trace is generated, an example of which can be seen in Table 8.

Index	Transition
1	`start_xtl_system`
2	`create(1, Init, [Init-alice, Recp-bob])`
3	`create(2, Recp, [Init-alice, Recp-bob])`
4	`start`
5	`gen(1, key, fresh(1))`
6	`send(1, [pk(bob), pk(alice), enc(pk(alice), enc(sk(bob), [pk(alice), fresh(1)]))])`
7	`recv(2, [pk(bob), pk(alice), enc(pk(alice), enc(sk(bob), [pk(alice), fresh(1)]))])`

Table 8: Example trace of XTL model generated from Listing 10

6.3 *lisb*-Backend (*dy2lisb*)

The Prolog translation shown in Section 6.2.3 closely follows the term-based model defined in Section 4. This is possible because the Prolog language provides the ability to freely generate nested terms and to mix predicates and actions while still backtracking when an inner predicate fails. Despite the B language offering set theory to easily replicate mathematical models, its additional features are distinctly lacking. Sequentially executing interleaved actions and predicates is difficult and recursive data structures require freetypes, which lack compatibility with other parts of the ProB toolchain. This makes a straightforward translation, as demonstrated in Section 6.2.3, infeasible and forces exploration of workarounds for those quirks. Additionally, due to the support for multiple output formats, the *dy2lisb* backend can be configured with multiple flags to control the translation and to make it fit the desired output format.

6.3.1 B representation of the Dolev-Yao model

As the B language requires that every identifier is statically typed we first define the types used in the model, as shown in Listing 25. The `SETS` clause is used to define new enumeration types, while the `FREETYPES` clause is used to define freetypes, which are further explained in Section 6.3.2. Like in the XTL representation, there is the *phase* type and a listing of all *agents*, *roles* and *variables* used in the protocol. But the action list is something new. In the XTL implementation, the protocol definition contains terms describing the actions, like `send`, and their arguments, for example the message pattern to be sent or received. This is not easily replicated in B, as we cannot implement a generic operation similar to the recursive `build_term_from_pattern` predicate. Thus each action is translated as a separate B operation, which has the parameters given in the protocol description built into its implementation. To distinguish between each action they are numbered, so the action sequence `generate`, `send`, `receive` of a protocol role `Init` would become `Init_1_generate`,

`Init_2_send` and `Init_3_receive`.

Listing 25: Types in the B machine for the `visualization` preset

```
SETS
    PHASE = {Create, Run, Done};
    AGENT = {Alice, Bob, Compromised1, Compromised2};
    /* these are protocol dependent: */
    ROLE = {Init, Recp};
    ACTION = {Init_1_generate, Init_2_send,
              Recp_1_receive, ThreadDone};
    VARIABLE = {Init_key, Recp_x}
FREETYPES
    TERM = PublicKey(AGENT),
           SecretKey(AGENT),
           Key(AGENT*AGENT),
           Enc(struct(key: TERM, term: TERM)),
           Seq(seq(TERM)),
           Fresh(NATURAL1),
           Hash(TERM)
```

The `CONSTANTS` clause contains the protocol, the maximum thread count, the set of compromised agents, the inverse key function and the set of valid role-to-agent mappings for thread creation. The role-to-agent mappings are written down explicitly to reduce the number of equivalent threads, es explained in Section 9.2. When declaring constants, it is also required to add a `PROPERTIES` clause, which types all constants and gives them values. In B this looks like Listing 26 for the example protocol from Listing 10.

Listing 26: Constants in the B machine for the `visualization` preset

```
CONSTANTS
    MaxThreads,
    Protocol,
    InverseKey,
    CompromisedAgents,
    ValidSubs
PROPERTIES
    MaxThreads=2 &
    Protocol={
        (Init, [Init_1_generate, Init_2_send]),
        (Recp, [Recp_1_receive])
    } &
    InverseKey=
        %(k).(k:ran(PublicKey)|SecretKey((PublicKey~)(k))) \/
        %(k).(k:ran(SecretKey)|PublicKey((SecretKey~)(k))) \/
        %(k).(k:ran(Key)|k) &
    CompromisedAgents={Compromised1, Compromised2} &
    ValidSubs={
        {(Init, Alice), (Recp, Bob)},
```

```
20:         {(Init, Alice), (Recp, Compromised1)},
21:         {(Init, Compromised1), (Recp, Alice)},
22:         {(Init, Compromised1), (Recp, Compromised2)}
23:     }
```

The state variables are identical to the data saved in the XTL state, but structured a bit differently. Instead of one big list containing all thread data in one term, the B representation splits them up into different variables, each containing a part of the data in a function indexed by the thread identifier. This simplifies the access patterns and also separates variables that rarely change from the ones that have to change a lot, thus helping ProB's internal optimizations. The definitions are shown in Listing 27. All state variables are listed in Table 9.

Listing 27: Variables in the B machine for the `visualization` preset

```
1:  VARIABLES
2:      phase,
3:      threads,
4:      thread_actions,
5:      thread_memory,
6:      adversary_knowledge,
7:      next_fresh
8:  INVARIANT
9:      phase:PHASE &
10:     threads:seq(struct(role:dom(Protocol), sub:dom(Protocol)+->AGENT)) &
11:     thread_actions:dom(threads)-->seq(ACTION) &
12:     thread_memory:dom(threads)-->(VARIABLE+->TERM) &
13:     adversary_knowledge<:TERM &
14:     next_fresh:NATURAL1
15: INITIALISATION
16:     phase := Create ||
17:     threads := {} ||
18:     thread_actions := {} ||
19:     thread_memory := {} ||
20:     adversary_knowledge := PublicKey[AGENT] \/
21:                            SecretKey[CompromisedAgents] \/
22:                            Key[CompromisedAgents*AGENT] \/
23:                            Key[AGENT*CompromisedAgents] ||
24:     next_fresh := 1
```

B state variable	Description
`phase`	Phase of protocol execution, either `Create`, `Run` or `Done`
`threads`	The starting parameters of the currently running threads
`thread_actions`	The list of actions for each thread that need to be executed
`thread_memory`	Thread-local memory of each thread
`adversary_knowledge`	List of terms that are known to the adversary
`next_fresh`	Counter to help generate unique terms

Table 9: B state variables for the `visualization` preset

The *phase* variable states the current phase of protocol execution and either allows thread creation or action execution. *threads* contains the list of the starting parameters of each thread. The index in this sequence determines the numerical thread identifier of each thread. *thread_ actions* and *thread_ memory* save the runtime state of each state, this being its action sequence and its memory respectively. The *adversary_ knowledge* set keeps tracks of all terms that the adversary managed to collect during execution. It starts out containing all public keys and the secret and shared keys of all compromised agents. Similar to the XTL state, the *next_fresh* variable is a global counter used to generate unique fresh values.

Operations can be separated into static ones, that are always the same for all protocols, and the protocol dependent ones, that are generated especially for the protocol at hand. The static operations are the lifecycle operations `create`, which creates a new thread, `start`, which transitions into the `Run` phase, `finish_thread` to mark the termination of a thread, and `finish`, which marks the whole protocol execution as `Done`. The implementations are shown in Listing 28.

Listing 28: Static operations in the B machine for the `visualization` preset

```
create(tid, role, sub) =
    SELECT phase=Create &
        tid=size(threads)+1 &
        tid<=MaxThreads &
        role:dom(Protocol) &
        sub:ValidSubs
    THEN
        threads(tid) := rec(role: role, sub: sub) ||
        thread_actions(tid) := Protocol(role) <- ThreadDone ||
        thread_memory(tid) := {}
    END;
start =
    SELECT phase=Create THEN
        phase := Run
    END;
finish =
    SELECT phase=Run &
        ran(thread_actions)={[]}
    THEN
```

```
20:         phase := Done
21:     END;
22: finish_thread(tid) =
23:     SELECT phase=Run &
24:         tid:dom(threads) &
25:         {ThreadDone}=thread_actions(tid)[{1}]
26:     THEN
27:         thread_actions(tid) := []
28:     END
```

The final static operations are the `unpack` and `decrypt` operations, which have no counterpart in the XTL representation, and allow the adversary to unpack `Seq` terms and to decrypt `Enc` terms using their keys – implementing one half of the term derivation relation. The other half is implemented in the `receive` operations. This split is necessary, because checking if a given term can be derived from the current adversary knowledge is a recursive operation for the given term, and for each term contained in the adversary knowledge. So the work is separated, each term added to the adversary knowledge is split as far as possible into its smallest constituents and when generating a message term to receive, only this message has to be broken down for checking of derivability, not each term in the adversary knowledge. The implementations are shown in Listing 29.

Listing 29: Static term inference operations in the B machine for the `visualization` preset

```
1: unpack(term, unpacked) =
2:     SELECT phase=Run &
3:         term:adversary_knowledge &
4:         term:ran(Seq) &
5:         unpacked=ran((Seq~)(term))\adversary_knowledge &
6:         unpacked/={}
7:     THEN
8:         adversary_knowledge := adversary_knowledge\/unpacked
9:     END;
10: decrypt(term, key, decrypted) =
11:     SELECT phase=Run &
12:         term:adversary_knowledge &
13:         term:ran(Enc) &
14:         decrypted=(Enc~)(term)'term &
15:         LET enc_key BE
16:             enc_key=(Enc~)(term)'key
17:         IN
18:             enc_key:dom(InverseKey) &
19:             key=InverseKey(enc_key) &
20:             key:adversary_knowledge
21:         END &
22:         decrypted/:adversary_knowledge
23:     THEN
24:         adversary_knowledge := adversary_knowledge\/{decrypted}
25:     END
```

While the XTL operations are generic and the message patterns are passed via the actions, each protocol action for each role is translated into a unique operation which has the message pattern and the generation of the message term encoded into it. This is again caused by the difficulty of implementing the derivation of the message pattern into possibly multiple valid message terms. The `generate` operations work similar to the XTL implementations: pop the action from the `thread_actions` list, make a new term of type `Fresh`, increment `next_fresh` and save it to `thread_memory` under the matching key. The `send` operations creates the message term directly and adds it to the adversary knowledge after popping the action value. Both implementations can be seen in Listing 30.

Listing 30: Exemplary protocol operations in the B machine for the `visualization` preset

```
Init_1_generate(tid, var, term) =
    SELECT phase=Run &
        {Init_1_generate}=thread_actions(tid)[{1}] &
        tid:dom(threads) &
        var=Init_key &
        term=Fresh(next_fresh)
    THEN
        thread_actions(tid) := tail(thread_actions(tid)) ||
        thread_memory(tid)(var) := term ||
        next_fresh := next_fresh+1
    END;
Init_2_send(tid, term) =
    SELECT phase=Run &
        {Init_2_send}=thread_actions(tid)[{1}] &
        tid:dom(threads) &
        term=Seq([
            PublicKey(threads(tid)'sub(Init)),
            PublicKey(threads(tid)'sub(Recp)),
            Enc(rec(
                key: PublicKey(threads(tid)'sub(Recp)),
                term: Enc(rec(
                    key: SecretKey(threads(tid)'sub(Init)),
                    term: Seq([
                        PublicKey(threads(tid)'sub(Recp)),
                        thread_memory(tid)(Init_key)
                    ])
                ))
            ))
        ])
    THEN
        thread_actions(tid) := tail(thread_actions(tid)) ||
        adversary_knowledge := adversary_knowledge\/{term}
    END
```

In contrast to the previous action implementations, the `receive` operations are extensive. After popping the action value, the message to be received is built up, subterm by subterm,

from the inside out. This uses nested `LET` substitutions for each term and `ANY` substitutions for nondeterminism to mimic Prolog's backtracking. Nondeterminism is required for variables in the message pattern: they could be set to any existing fresh value or the adversary could generate a new fresh value. But the act of generating a new fresh value requires changing the current state, as the global counter *next_fresh* needs to be updated and the newly generated value needs to be added to the *adversary_knowledge*. The following substitutions need to use the updated state variables, and so all the following subterm generators are nested inside a `LET` binding, which keeps track of the current transient state variables, whose changes are not yet committed. After all subterms have been generated, the execution worked through many layers of nested `LET` blocks and each subterm is saved in a binding called `termN`. The innermost action is a `SELECT` statement, which checks if the message term that was just generated is derivable by the adversary, before finally committing and setting the state variables. Selected parts of the implementation are shown in Listing 31.

Listing 31: Parts of a `receive` operation in the B machine for the `visualization` preset

```
/* creating a public key term */
LET term3,adversary_knowledge3,next_fresh3,thread_memory3 BE
    term3=PublicKey(threads(tid)'sub(Recp)) &
    adversary_knowledge3=adversary_knowledge2 &
    next_fresh3=next_fresh2 &
    thread_memory3=thread_memory2
IN ... END

/* nondetermism to choose a fresh value for a variable */
ANY create_new6 WHERE create_new6:BOOL THEN
    ANY term6 WHERE
        term6:IF create_new6=TRUE THEN
            {Fresh(next_fresh5)}
        ELSE
            {t|t:ran(Fresh) &
               (Fresh~)(t):1..next_fresh4}
        END
    THEN
        LET adversary_knowledge6,next_fresh6,thread_memory6 BE
            adversary_knowledge6=IF create_new6=TRUE THEN
                adversary_knowledge5\/{term6}
            ELSE
                adversary_knowledge5
            END &
            next_fresh6=IF create_new6=TRUE THEN
                next_fresh5+1
            ELSE
                next_fresh5
            END &
            thread_memory6=thread_memory5 <+
                           {(tid, thread_memory5(tid) <+
                                  {(Recp_x, term6)})}
```

```
33:         IN ... END
34:     END
35: END
36:
37: /* committing to the generated term */
38: SELECT term10:adversary_knowledge10 or
39:     (term1:adversary_knowledge10 &
40:     term2:adversary_knowledge10 &
41:     (term9:adversary_knowledge10 or
42:     (term3:adversary_knowledge10 &
43:     (term8:adversary_knowledge10 or
44:     (term4:adversary_knowledge10 &
45:     (term7:adversary_knowledge10 or
46:     (term5:adversary_knowledge10 &
47:     term6:adversary_knowledge10))))))) &
48:     term=term10
49: THEN
50:     adversary_knowledge := adversary_knowledge10 ||
51:     next_fresh := next_fresh10 ||
52:     thread_memory := thread_memory10
53: END
```

Instead of using nested `LET` statements, one could replace them with a `VAR` block that defines each `termN` variable and contains sequential compositions to do each nested operation in sequence. This however would cause problems with the TLA$^+$ translation, as sequential statements are not supported in TLA$^+$. Each TLA$^+$ operation is effectively a single before-after-predicate, and so the sequential substitutions have to be rewritten to parallel substitutions by TLC4B anyway. But the algorithm applied in TLC4B is quite brittle and manually implementing the translation saves the trouble.

When animating the B model a trace is generated, an example of which can be seen in Table 10.

Index	Transition
1	SETUP_CONSTANTS
2	INITIALISATION
3	create(tid=1, role=Init, sub=(Init|->Alice), (Recp|->Bob))
4	create(tid=2, role=Recp, sub=(Init|->Alice), (Recp|->Bob))
5	start
6	Init_1_generate(tid=1, var=Init_key, term=Fresh(1))
7	Init_2_send(tid=1, term=Seq((1|->PublicKey(Alice)), (2|->PublicKey(Bob)), (3|->Enc(rec(key:PublicKey(Bob), term:Enc(rec(key:SecretKey(Alice), term:Seq((1|->PublicKey(Bob)), (2|->Fresh(1))))))))))
8	finish_thread(tid=1)
9	Recp_1_receive(tid=2, term=Seq((1|->PublicKey(Alice)), (2|->PublicKey(Bob)), (3|->Enc(rec(key:PublicKey(Bob), term:Enc(rec(key:SecretKey(Alice), term:Seq((1|->PublicKey(Bob)), (2|->Fresh(1))))))))))
10	finish_thread(tid=2)
11	finish

Table 10: Example trace of B model generated from Listing 10

6.3.2 Freetypes and alternative representation

Classical B does not support freetypes and Event-B supports recursive or inductive datatypes only via the theory plugin. But the ProB flavor of classical B adds support for them[4]. Freetypes are declared in a new section, fittingly called `FREETYPES`. Each freetype definition has a name and contains a list of *constructors* that are either blank identifiers or contain a set denoting the parameter type in parentheses. An example definition can be seen in Listing 32.

Listing 32: Freetype to define a linked list in ProB

```
1: FREETYPES
2:     IntList = inil,
3:               icons(INTEGER*IntList)
```

When working with freetypes, the name identifier can be used like a set containing all possible values of the declared freetype for membership tests in typing predicates. Constructors represent the distinct subtypes of a freetype and are used to create a value of that subtype. In the semantic of B they are constant values if they have no parameter. Those constructors can be used as-is. When they do have parameters, they act like a total function from the parameter type of the constructor to the freetype. Calling such a constructor like a

[4]please refer to https://prob.hhu.de/w/index.php?title=Free_Types for more information

function, with a value of the declared parameter type, yields a freetype value containing the passed parameter value. Using the reverse operator ~ creates a function that can extract the parameter value from a freetype value of that subtype. A usage example of the linked list freetype looks like Listing 33.

Listing 33: Using the linked list freetype from Listing 32

```
VARIABLES l
INVARIANT l : IntList
INITIALISATION l := inil
OPERATIONS
    add(i) = SELECT i : INTEGER THEN l := icons(i, l) END;
    pop = SELECT l : ran(icons) THEN l := prj2(icons~(l)) END;
    i <-- peek = SELECT l : ran(icons) THEN l := prj1(icons~(l)) END
```

But union types and recursive types are not supported by Event-B or TLA$^+$, thus an alternative representation is needed. The general idea is to save all freetype values ever created in a global table and all references to that value use its table index. This only works if all freetype constructor parameter types are compatible with `INTEGER`, which is the case for our linked list example. A rewrite of the prior example can be seen in Listing 34.

Listing 34: Implementing the linked-list freetype using classical B

```
SETS IntListType = {inil, icons}
VARIABLES IntList, l
INVARIANT
    IntList : seq(IntListType*seq(INTEGER)) &
    !(fv).(fv : ran(IntList) =>
        (fv = (inil, []) or
         (prj1(fv)=icons & size(prj2(fv))=2 &
          prj2(fv)(2) : dom(IntList)))) &
    l : dom(IntList)
INITIALISATION IntList := [(inil, [])]; l := 1
OPERATIONS
    add(i) = SELECT i : INTEGER THEN
        IntList := IntList<-(icons, [i, l]);
        l := size(IntList)
    END;
    pop = SELECT prj1(IntList(l)) = icons THEN
        l := prj2(IntList(l))(2)
    END;
    i <-- SELECT prj1(IntList(l)) = icons THEN
        i := prj2(IntList(l))(1)
    END
```

But this representation has a problem, no value added to the global table is ever removed. This is a memory leak and can be mitigated partially by checking if a value already exists in the table before adding a new entry, as demonstrated in Listing 35. However, the approach only takes care of not adding duplicate values, but unused values are still never deleted,

which would require an implementation of automatic memory management like reference counting or garbage collection. Note how the required B code to create a freetype value got larger for just one layer of freetype, doing this for a deeply nested recursive freetype like a message term used in a `send` operation becomes rather excessive.

Listing 35: Partially mitigating the memory leak of Listing 34

```
1:  add(i) = SELECT i : INTEGER THEN
2:      LET fv BE fv = (icons, [i, l]) IN
3:          IF fv : ran(IntList) THEN
4:              l := IntList~(fv)
5:          ELSE
6:              IntList := IntList<-fv;
7:              l := size(IntList)
8:          END
9:      END
10: END
```

The *dy2lisb* backend uses freetypes to represent terms, which looks like Listing 36.

Listing 36: Freetype representation of Dolev-Yao terms

```
1:  SETS
2:      AGENT = {Alice, Bob, Compromised1, Compromised2, ...}
3:  FREETYPES
4:      TERM = PublicKey(AGENT),
5:             SecretKey(AGENT),
6:             Key(AGENT*AGENT),
7:             Enc(struct(key: TERM, term: TERM)),
8:             Seq(seq(TERM)),
9:             Fresh(NATURAL1),
10:            Hash(TERM)
```

But toggling the `:use-freetypes` option makes the translation compatible with Event-B and TLA$^+$ by using the representation in Listing 37. Note that the user-defined enumerated set `AGENT` got reduced to a set of integers. This is necessary to fit all data into a sequence of integers, but reduces the comprehension of term values for users. Additionally all keys have to be precomputed because they are needed in the `Properties` clause to define the `InverseKey` function.

Listing 37: Freetype representation of Dolev-Yao terms

```
1:  SETS
2:      TERM = {PublicKey, SecretKey, Key, Enc, Seq, Fresh, Hash}
3:  CONSTANTS AGENT
4:  PROPERTIES
5:      AGENT = {1, 2, 3, 4, ...}
6:  VARIABLES all_terms
7:  INVARIANT
8:      all_term : seq(TERM*seq(NATURAL1))
```

6.3.3 Records and alternative representation

Record, or equivalently structs, are a way to group data of different types together. Each entry has a distinct name, which is needed to access it from the record value. As opposed to sets and freetypes, a record does not have to be declared in a separate section of the machine. They can just be used in predicates and expressions. To create a new record, one has to use the syntax `rec(<entry1>:<value1>, ...)`, and the set of all records with a specific signature can be written as `struct(<entry1>:<type1>, ...)`. The entries of a record can be accessed with the single quote and the entry name, which looks like `r'e`. An example can be seen in Listing 38.

Listing 38: Using records in ProB

```
1:  VARIABLES x
2:  INVARIANT x : struct(a:INTEGER, b:BOOL, c:POW(INTEGER))
3:  INITIALISATION x := rec(a:0, b:FALSE, c:{})
4:  OPERATIONS
5:      set_a(i) = SELECT i : INTEGER THEN x'a := i END;
6:      toggle_b = BEGIN x'b := {(TRUE,FALSE),(FALSE,TRUE)}(x'b) END;
7:      add_c(i) = SELECT i : INTEGER THEN x'c := x'c \/ {i} END
```

However, Event-B does not support records. But an equivalent representation exists using tuples or nested pairs, as records and tuples are just ordered lists of heterogeneous values. The tuple `(x1, x2, ..., xN)` is represented internally as `((x1, ..., xN-1), xN)`, which is represented internally as `(((x1, ..., xN-2), xN-1), xN)` and so on. Thus accessing the values inside becomes harder, as one must use the `prj1` and `prj2` functions to navigate the nested pairs, and it is impossible to override parts of the tuple with a special syntax like for records. The last value in the tuple can be accessed with `prj2`, but the first one has to be accessed with $n-1$ nested `prj1` calls, where n is the size of the tuple. This is demonstrated in Listing 39.

Listing 39: Using tuples to represent the record in Listing 38

```
1:  VARIABLES x
2:  INVARIANT x : INTEGER*BOOL*POW(INTEGER)
3:  INITIALISATION x := (0, FALSE, {})
4:  OPERATIONS
5:      set_a(i) = SELECT i : INTEGER THEN
6:          x := (i, prj2(prj1(x)), prj2(x))
7:      END;
8:      toggle_b = LET binv BE
9:          binv = {(TRUE,FALSE),(FALSE,TRUE)}(prj2(prj1(x)))
10:     IN
11:         x := (prj1(prj1(x)), binv, prj2(x))
12:     END;
13:     add_c(i) = SELECT i : INTEGER THEN
14:         x := (prj1(prj1(x)), prj2(prj1(x)), prj2(x) \/ {i})
15:     END
```

The *dy2lisb* backend uses records by default for the implementation of encrypted messages and the thread initialization parameters. The option `:use-records` can be used to disable this feature and to replace all usages with tuples.

6.3.4 Other feature flags

The *dy2lisb* backend supports even more flags to change the translation behavior.

`:use-sequences` ProB supports sequences, which are a special type of function from the natural numbers to some range. The domain is always a finite interval $1..n$, so they work like arrays or lists in other programming languages. They can be written as a sequence literal with square brackets: `[1, 2, 3]`. Event-B lacks support for sequences, but *lisb*'s translation to Event-B can replace all usages of them with special constructs, that mimic the semantic of ProB's sequences. If that is not satisfactory, they can also be turned off by disabling the `:use-sequences` flag. This will replace all usages of `seq(T)` with `NATURAL1 +-> T`, while ignoring the additional domain constraints.

`:use-infinite-sets` As ProB is a constraint solver and thus supports symbolic techniques, infinite sets are usually not a problem. But the TLC model checker for TLA$^+$ is sensitive to infinite sets and does not allow explicit set operations with them. If – like in this case – infinite sets cause problems they can be replaced with finite equivalents. This is important for the built-in sets `INTEGER` and `NATURAL`, which have finite counterparts `INT` and `NAT`. The preferences `MIN_INT` and `MAX_INT` control the size of those alternatives.

`:use-complex-lhs-assignment` This flag toggles whether the left-hand side of an assignment may use complex expressions to override specific function or record entries. ProB supports them even when they are nested, like in this assignment which shows up in the `generate` operation: `thread_memory(tid)(var) := value`. Event-B however, does not support them, and thus they need to be rewritten to use the functional override `<+`. The example now becomes `thread_memory := thread_memory <+ {(tid, thread_memory(tid) <+ {(var, value)})}`.

`:use-eventb-prj` Classical B's `prj1` and `prj2` functions are not generic, they need to be instantiated with the type of the tuple one is trying to unpack: `prj1(INTEGER, BOOL)((1, TRUE))`. This is very verbose and thus Event-B made the tuple projections generic, which allows just passing the tuple value. ProB implements such generic projections as well, and the corresponding *dy2lisb* option is enabled by default.

`:use-if-pred` The `IF-THEN-ELSE` construct can be used in multiple contexts in ProB: in an operation as a substitution, and as an expression or predicate that returns values. But they are not natively supported by Event-B. Especially `IF`-predicates break some assumptions made in the ProB toolchain and do not work with Event-B, thus they are disabled by default and replaced with `IF`-expressions instead.

`:visualization` Enabling the `visualization` flag adds the definitions required to generate the UML sequence chart, as explained in Section 7.3.2. Those definitions get all their data about the state of the model from operation calls and their parameters, so additional parameters are added to existing operations and some new operations are added that are used to mark the completion of a thread.

6.3.5 The *dy2lisb* presets

`visualization` The `visualization` preset's goal is to allow for easy visualization and manual animation. Thus, the `:visualization` flag is enabled to allow generation of the UML sequence chart and the `:use-freetypes` and `:use-records` flags are enabled as well to allow the user to better comprehend the state of the model while animating.

`b` The `b` preset is aimed at compatibility with the ProB toolchain. It does not use freetypes or records.

`tla` TLA$^+$ supports records natively, thus this preset extends the `b` preset with the `:use-records` flag enabled.

`eventb` This preset is used together with the `eventb` output mode and copies the `b` preset, while disabling the `:use-complex-lhs-assignment` flag.

7 Applying the ProB toolchain

7.1 Translating classical B to Event-B

lisb is capable of outputting an Event-B project in addition to classical B [AK24]. This can be used by setting the `eventb` output mode when using *dy2prob*. An overview of the translation process can be seen in Figure 4.

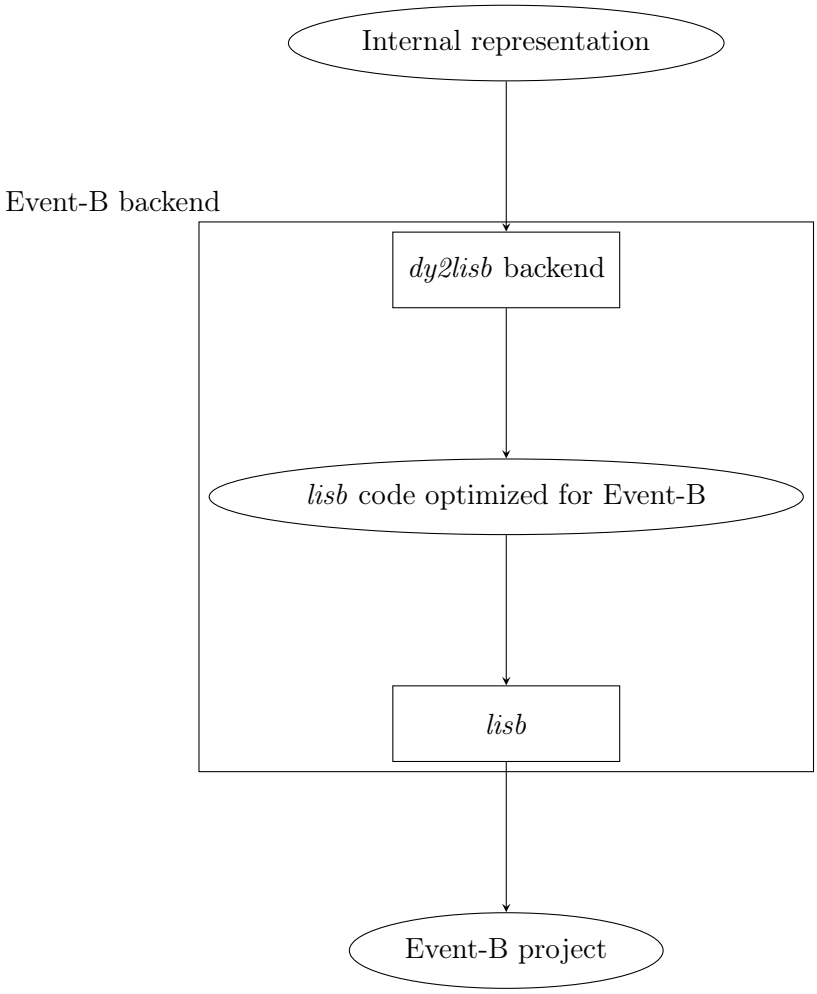

Figure 4: Event-B translation process

lisb will export an Event-B machine and a context. The machine contains all state variables and operations, while the context contains all sets and constants. As the Event-B language has fewer features than the ProB flavor of classical B, the representation for compatibility – no records and no freetypes – has to be used as the source for the translation. The missing features are freetypes, records and nested assignments for functions. While freetypes could

be implemented with the Event-B theory plugin [BM13], there was no *lisb* support for interfacing with the Event-B theory plugin at implementation time.

But the biggest difference compared to classical B is the non-existence of `IF` and `LET` constructs. All `LET`s can be implemented by syntactically replacing all usages of the declared local variables with their value, which increases the size of the nested expression or statement when the local variable is used multiple times. For the `IF` construct there are multiple possible translations depending on the context. An `IF` predicate can be replaced with a logical formula: `IF P THEN P1 ELSE P2 END` becomes `P => P1 & not(P) => P2`, whereas an `IF` expression can be implemented with a trick utilizing lambdas: `IF P THEN E1 ELSE E2 END` becomes `(%(t).(t : {true} & P | E1) \/ %(t).(t : {true} & not(P) | E2))(true)`. When an `IF P THEN S1 ELSE S2 END` construct is used in a substitution, the containing event has to be cloned, with one event adding the condition `P` into its guard and replacing the `IF` with the `S1` substitution, and the other one adding `not(P)` into its guard and replacing itself with the `S2` substitution or a `skip` if the `IF` had no `ELSE` clause. This also has the potential to blow up the size of an Event-B machine when many `IF` constructs are used.

To access the generated Event-B projects one needs to use Rodin. Rodin has no support for animation or model checking by default, but using the ProB Rodin plugin adds the ProB interface into Rodin and allows model checking and animating the machine, as demonstrated in Figure 5. Rodin intends for users to use its interactive proof system to validate the model, and a lot of proof obligations are generated for the Dolev-Yao model, although most of them do not pass by default. Trying to manually prove them is very difficult however, the invariants and guards need to be strengthened.

It is also possible to use the ProB Java API to import the generated Event-B model directly, which is used for testing of the *dy2lisb* backend.

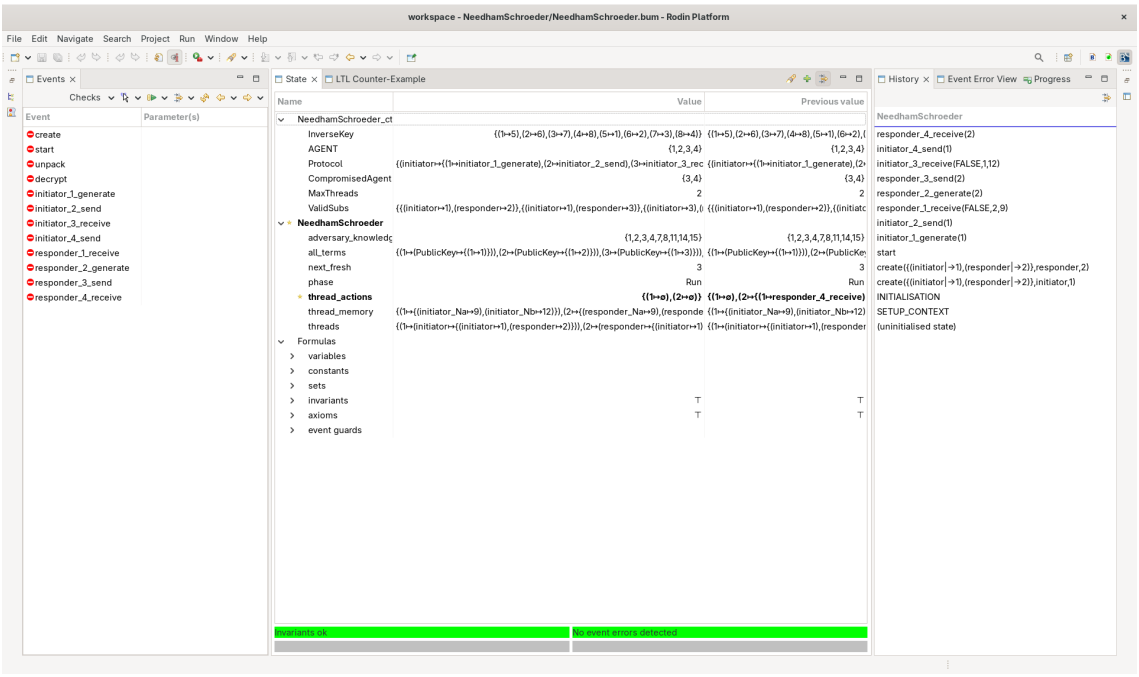

Figure 5: Animating an Event-B model with Rodin and the ProB plugin

7.2 Translating classical B to TLA+

The classical B representation can be translated into TLA+ using the TLC4B [HL14] library, which is implemented into ProB2-UI and into ProB Tcl/Tk, and then model checked with TLC. As the TLA+ language supports records natively, there exists a `tla` preset which keeps those enabled for a performance improvement when model checking. An overview of the translation process can be seen in Figure 6.

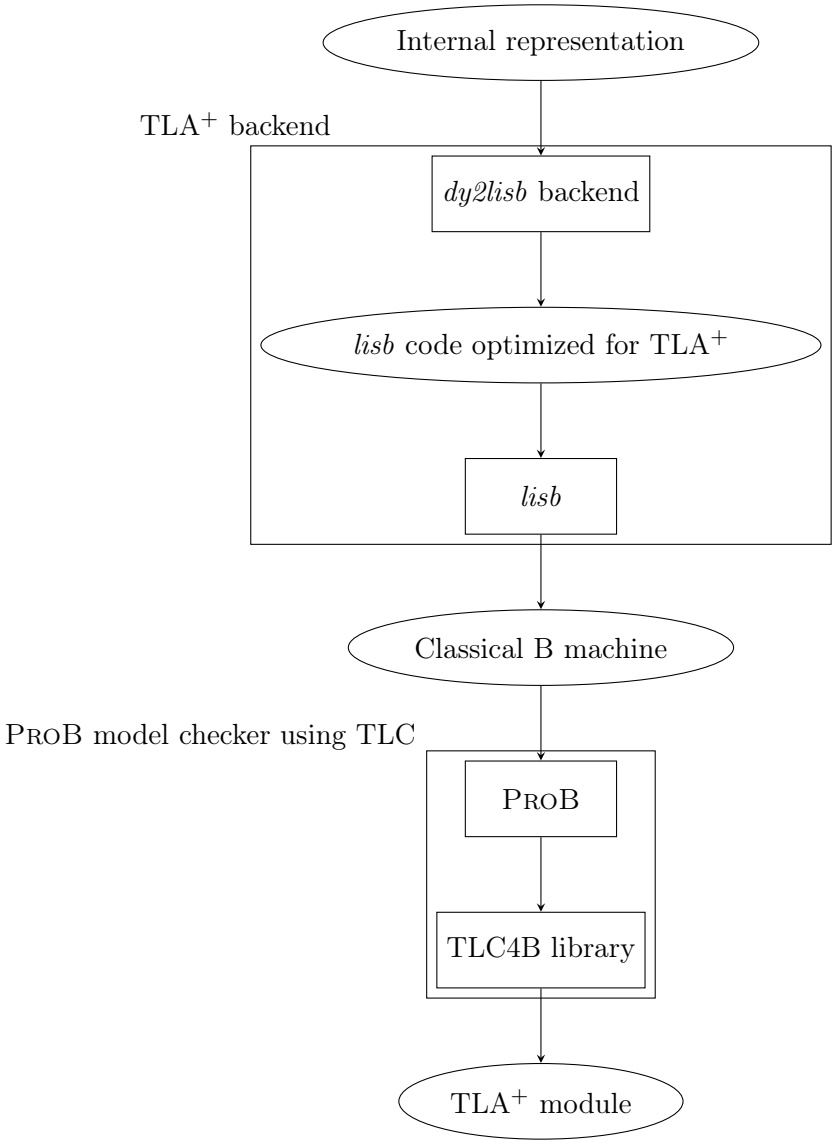

Figure 6: TLA$^+$ translation process

When using ProB2-UI it is possible to select the output path of the generated TLA$^+$ module and thus preserve it for manual checking afterwards. For this feature to work, one has to create a new TLC model checking task with the *Save generated files* option enabled, and select a folder with the path input next to it. This and the general user interface can be seen in Figure 7.

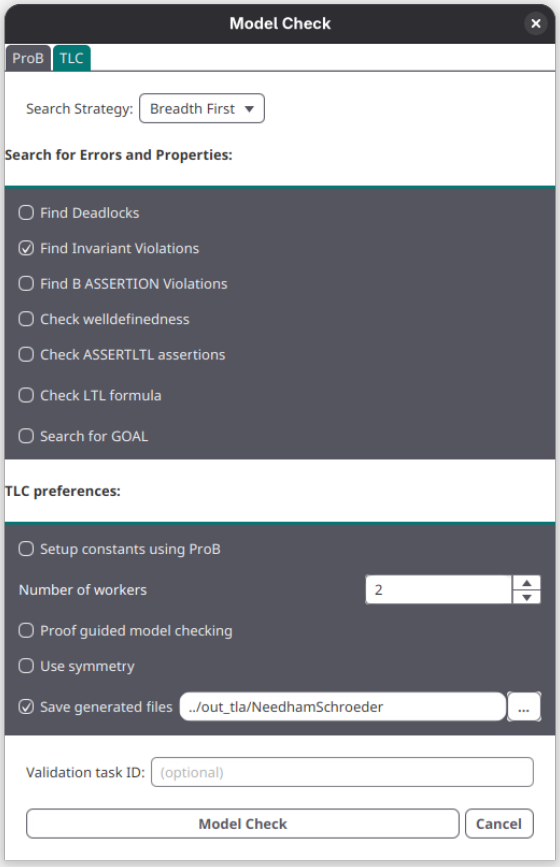

Figure 7: TLC model checking with ProB2-UI

When saving the generated files it becomes possible to inspect and model check the TLA+ module. An exemplary TLC model check can be seen in Figure 8.

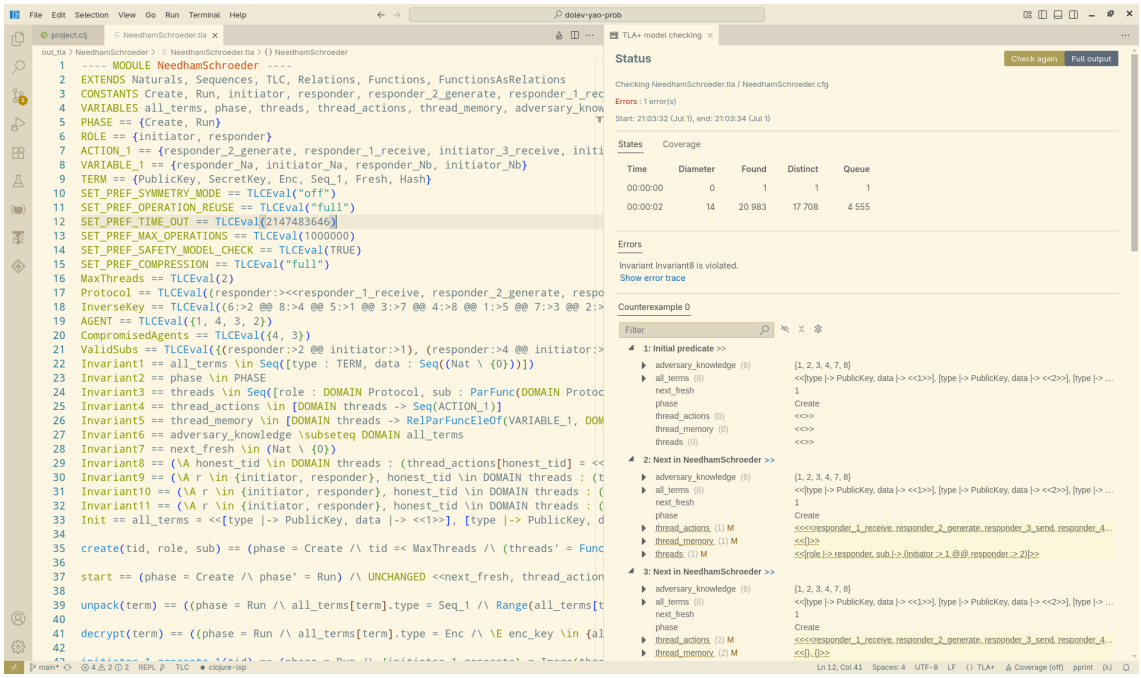

Figure 8: TLC model checking of the generated TLA$^+$ module

7.3 Animation and Visualization

Animation and visualization are two techniques usable in ProB to allow a better understanding and manual verification of formal models. Especially the classical B representation generated using the `visualization` preset is suitable for animation and visualization using the two ProB UIs. Additionally, a trace can be visualized as a UML sequence diagram, with special support to show the communication between the agents and the adversary.

7.3.1 Animation

While explicit state model checkers verify the whole state-space and produce a result, an animator allows the user to manually explore the state-space by selecting a transition from all that are possible in each state and allowing deeper inspection of all variables and constants. A user can verify if a specific trace exists or if all expected transitions are available in some state.

ProB2-UI allows all this and more: it is possible to do random animation or to execute a transition given a predicate for the resulting state or parameters. The user interface is shown in Figure 9.

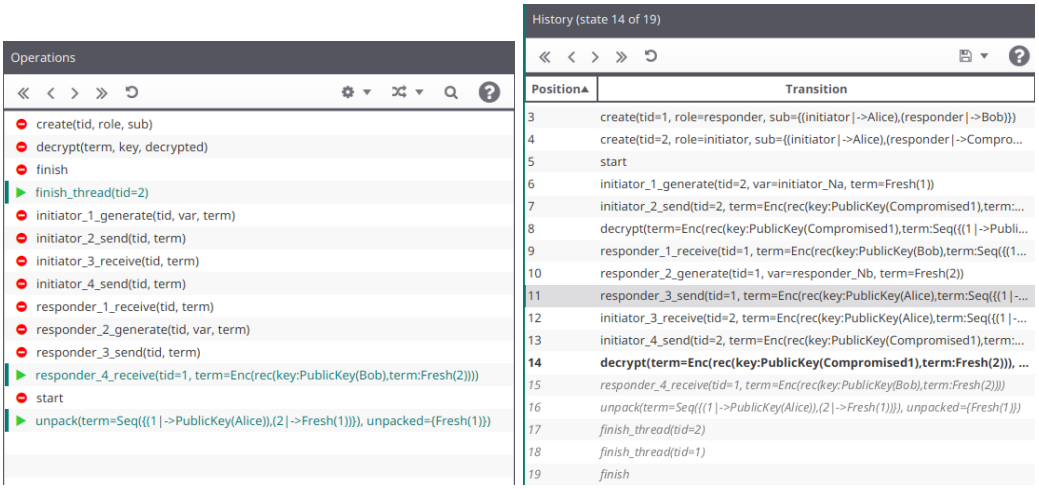

Figure 9: Animation and history view in ProB2-UI

The operations view on the left side lists all known transitions with their current status. If they are disabled – their guard predicate evaluates to `false` – they are shown with a red blocker icon. When there exists an assignment of parameters to satisfy the guard predicate, all computed parameter assignments are shown and the operation has a green play button as an icon. The desired transition can then be executed by clicking on it.

7.3.2 Visualization of traces using UML sequence charts

Another tool to help manual inspection of model properties is visualization. It entails the generation of a graphical image to explain the current state of the system. It even allows the user to interact with it by clicking, an idea which is explored by VisB [WL20]. But it is also possible to visualize a whole trace, for example using UML sequence charts generated by *plantuml*[5]. ProB contains functionality to generate a UML sequence chart using special entries in the `DEFINITIONS` section of the B machine. A visualization of the example protocol Listing 10 can be seen in Figure 10. This graphic will update dynamically when animating the model.

[5] https://plantuml.com/

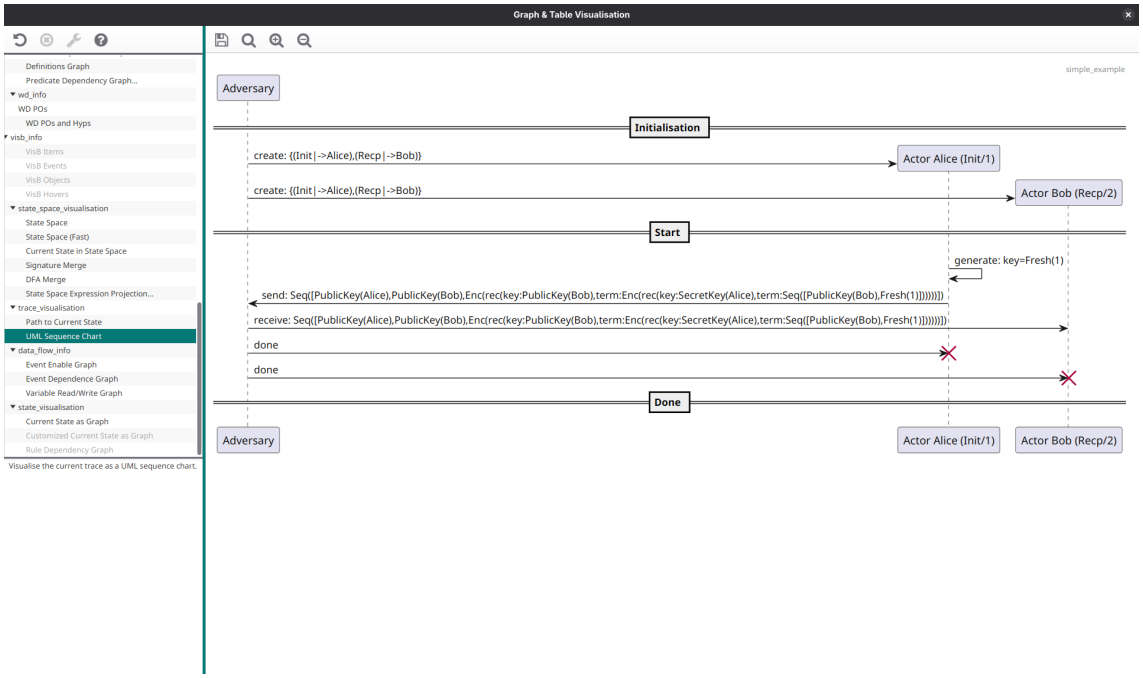

Figure 10: UML sequence chart in ProB2-UI

To configure this visualization, a definition entry needs to be created for each transition. The entry must have the form `SEQUENCE_CHART_<operation name>(<args>) == <value>`, where *value* can be a string in *plantuml* syntax[6] or structured data in the form of a tuple or record that configure the generated text and arrows. When generating a sequence chart, ProB will examine the history up to the current state and generate a line of *plantuml* description for each transition in it, which gets send to the *plantuml* application for rendering. A more extensive documentation is available in the ProB Wiki[7].

An example can be seen in Listing 40. This makes extensive use of the external functions `TO_STRING` and `FORMAT_TO_STRING` to build the desired strings at runtime from the operation parameters. ProB supports template strings which would simplify this code, but *lisb* does not yet support them at this time.

Listing 40: UML sequence chart definitions generated for the protocol in Listing 10

```
1: SEQUENCE_CHART_SETUP_CONSTANTS == "header simple_example";
2: SEQUENCE_CHART_INITIALISATION == "participant \"Adversary\" as
       Adversary\n== Initialisation ==";
3: SEQUENCE_CHART_create(tid, role, sub) ==
       FORMAT_TO_STRING("Adversary -> \"~w\" ** : create: ~w",
       [FORMAT_TO_STRING("Actor ~w (~w/~w)", [TO_STRING(sub(role)),
       TO_STRING(role), TO_STRING(tid)]), TO_STRING(sub)]);
```

[6] https://plantuml.com/guide
[7] https://prob.hhu.de/w/index.php?title=Generating_UML_Sequence_Charts

```
4:  SEQUENCE_CHART_start == "== Start ==";
5:  SEQUENCE_CHART_finish == "== Done ==";
6:  SEQUENCE_CHART_finish_thread(tid) == FORMAT_TO_STRING("Adversary ->
        \"~w\" !! : done", [FORMAT_TO_STRING("Actor ~w (~w/~w)",
      [TO_STRING(threads(tid)'sub(threads(tid)'role)),
      TO_STRING(threads(tid)'role), TO_STRING(tid)])]);
7:  SEQUENCE_CHART_unpack(term, unpacked) ==
      FORMAT_TO_STRING("Adversary --> Adversary : unpack: ~w\nnote
      left : New Adversary Knowledge:\\n~w", [TO_STRING(term),
      TO_STRING(unpacked)]);
8:  SEQUENCE_CHART_decrypt(term, key, decrypted) ==
      FORMAT_TO_STRING("Adversary --> Adversary : decrypt with ~w: ~
      w\nnote left : New Adversary Knowledge:\\n~w", [TO_STRING(key),
      TO_STRING(term), TO_STRING(decrypted)]);
9:  SEQUENCE_CHART_Init_1_generate(tid, var, term) ==
      FORMAT_TO_STRING("\"~w\" -> \"~w\" : generate: ~w=~w",
      [FORMAT_TO_STRING("Actor ~w (~w/~w)",
      [TO_STRING(threads(tid)'sub(threads(tid)'role)),
      TO_STRING(threads(tid)'role), TO_STRING(tid)]),
      FORMAT_TO_STRING("Actor ~w (~w/~w)",
      [TO_STRING(threads(tid)'sub(threads(tid)'role)),
      TO_STRING(threads(tid)'role), TO_STRING(tid)]), "key",
      TO_STRING(term)]);
10: SEQUENCE_CHART_Init_2_send(tid, term) == FORMAT_TO_STRING("\"~w\" -
      > Adversary : send: ~w", [FORMAT_TO_STRING("Actor ~w (~w/~w)",
      [TO_STRING(threads(tid)'sub(threads(tid)'role)),
      TO_STRING(threads(tid)'role), TO_STRING(tid)]),
      TO_STRING(term)]);
11: SEQUENCE_CHART_Recp_1_receive(tid, term) ==
      FORMAT_TO_STRING("Adversary -> \"~w\" : receive: ~w",
      [FORMAT_TO_STRING("Actor ~w (~w/~w)",
      [TO_STRING(threads(tid)'sub(threads(tid)'role)),
      TO_STRING(threads(tid)'role), TO_STRING(tid)]),
      TO_STRING(term)]);
```

Another feature for visualization is the HTML export functionality. It allows one to export a trace as an interactive HTML page, which supports replaying the trace while inspecting the visualization, as shown in Figure 11.

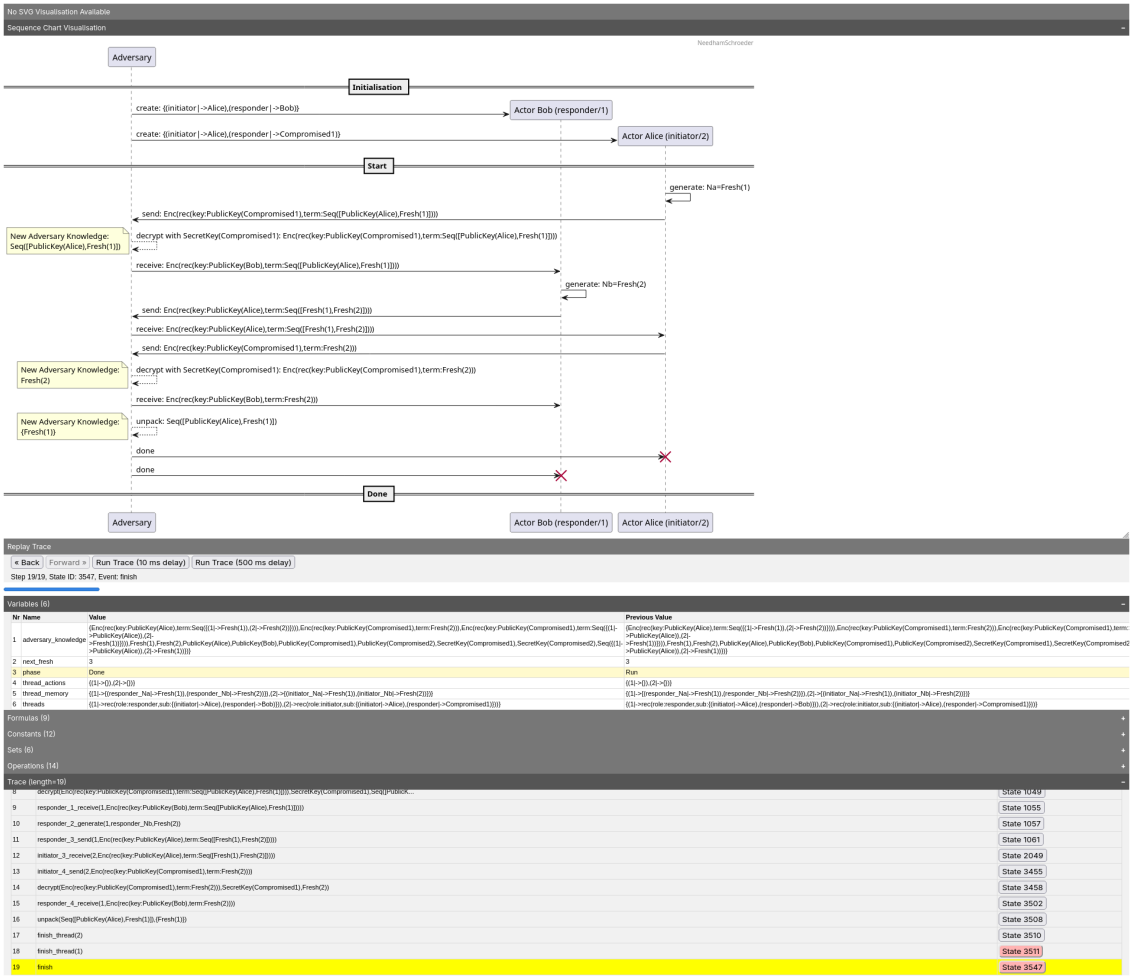

Figure 11: HTML trace export generated by ProB

7.4 Model checking

All generated models can be checked using the ProB explicit state model checker, and the B models can also be checked with the LTSmin [Kan+15] and TLC model checkers [Lam99]. As the protocol properties are encoded as state invariants, the model checker just needs to search for invariant violations, while ignoring deadlocks, as those mean that the protocol has finished running or can no longer progress. All the aforementioned checkers are explicit state model checkers, so they will explore the whole state-space to find all reachable states and check each of them individually for a violation of all required properties. If the state-space is finite they will give an answer, barring any memory issues, after a finite amount of time. Throughout the whole modeling process, attention has been paid to keep the state-space finite, even though the formal model allows infinite agents and the base sets are supposed to be infinite. Further explanation can be found in Section 9.3.

The LTSmin model checker is currently only accessible from ProB's Tcl/Tk user interface, while the other model checkers can be accessed from ProB2-UI as well. Figure 12 shows the ProB model checking interface which is used to configure model checking of the classical B, Event-B and XTL models, while Figure 7 shows the TLC model checker which can only be applied to the classical B model, preferably the one generated with the `tla` preset. To run a model checking job in ProB2-UI, one has to access the *Verifications* tab in the main view and then navigate to the *Model Checking* tab within it. There is a button, marked with an image of a plus, to add a new model checking task, which opens the configuration window shown in Figure 12. After adding the job, it can be saved in the current project and re-run when desired with the run button, marked by a checkmark. After the model checking task has finished, the result will be displayed in the *Status* column and some statistics are visible below. If a state was found that breaks a required property of the model, like an invariant violation or a deadlock, it can be inspected by clicking on the *Show trace* button in the errors table.

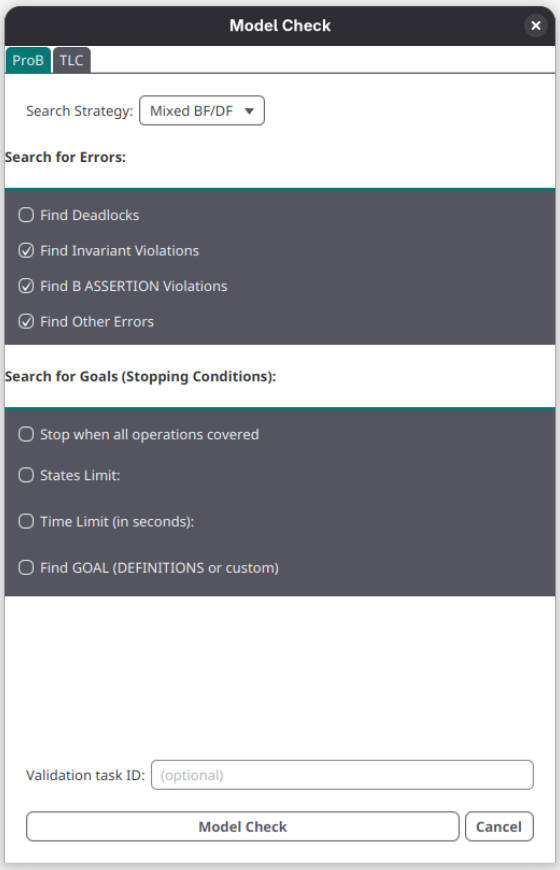

Figure 12: ProB model checking with ProB2-UI

8 Improving the ProB toolchain

In the course of implementing *dy2prob* according to the previous specifications, some roadblocks were encountered. Missing features had to be added to different parts of the ProB toolchain to make the B and XTL machine generation possible.

The heart of the ProB is ProB Core, the model checker, animator and constraint solver implemented in Prolog. To improve the animation of XTL and B models, the pretty-printer was improved, it will now always print XTL values as Prolog would, and even print B sets as sequences where possible. These changes are released as version 1.15.0 of the ProB CLI.

The parser library *probparsers* is implemented in Java and parses B machine text files into abstract syntax trees (AST). But it also has a pretty-printer for B which can generate a textual representation of an AST. Furthermore, it has a Prolog term reader and writer implementation, used to facilitate communication with the ProB core from the Java world. Here the pretty-printer was updated to support all syntax constructs, to improve indentations and to fix some whitespace issues, which allows *lisb* to print `LET`s and `IF`s in all contexts. The Prolog library was equipped with the ability to print Prolog lists written with a tail, for example `[H1, H2 | T]`. This was needed to generate the XTL representation. These changes are part of the 1.15.0 version of the ProB CLI and were released as version 2.15.2 of the *probparsers* project.

The ProB Java API sits on top of the ProB core and allows Java-to-Prolog interoperability. Using this API, ProB2-UI was built as a replacement for the older ProB Tcl/Tk editor. During the work on this thesis, the visualization functionality was made more generic to support the *plantuml* visualization for the UML sequence chart, which also required changes in the ProB Java API. These changes are part of the 1.15.0 version of the ProB CLI and were released as version 4.15.0 of the ProB Java API. There is currently no released version of ProB2-UI that contains all these changes, to obtain them is necessary to run the specific commit `4876b4d`[8] from source.

The TLC4B and TLA2BAST Java libraries are used for the translation of TLA$^+$ code to B and vice-versa. To allow TLC model checking of the generated Dolev-Yao B machine, TLC4B was modernized, similar to the *probparsers* pretty-printers, to support all `LET` and `IF` constructs. Additionally, the algorithm for functions on the left-hand side of assignments was improved to support nested functions and records. To allow the back-translation of the generated TLA$^+$ specification into B, the type inference of TLA2BAST was refactored and massively improved. These changes are released as version 1.4.2 of TLA2BAST and 1.2.2 of TLC4B, which are not part of the 1.15.0 version of the ProB CLI.

The biggest changes were made to *lisb*'s Event-B backend. On the one hand the `prj1` and `prj2` functions were not supported in their generic form for B machines and for back-translation, and on the other hand the Event-B backend lacked support for `IF` and `LET`

[8] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob2_ui/-/commit/4876b4d5ad7e7a75a48b82b419e753804c747b4e

constructs in all contexts. Both of these were implemented, which required a rewrite of the classical B to Event-B translation. These changes are not yet merged, but a pre-built version is available and used in the `lib` directory of the *dy2prob* repository[9].

Having a working implementation now, it is possible to compare the results of the automatic B and XTL machine generation with each other and with existing tools.

[9] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/vella-master-code

9 Evaluation

To set up comparable conditions for any of the following benchmarks, we use ProB2-UI and take the displayed model checking stats from the *Verifications* tab in ProB2-UI, as shown in Figure 13. All benchmarks were performed using an Intel i7-1165G7 with 16 GB RAM, the Linux operating system and the nightly version of the ProB toolchain as of 03.07.2025.

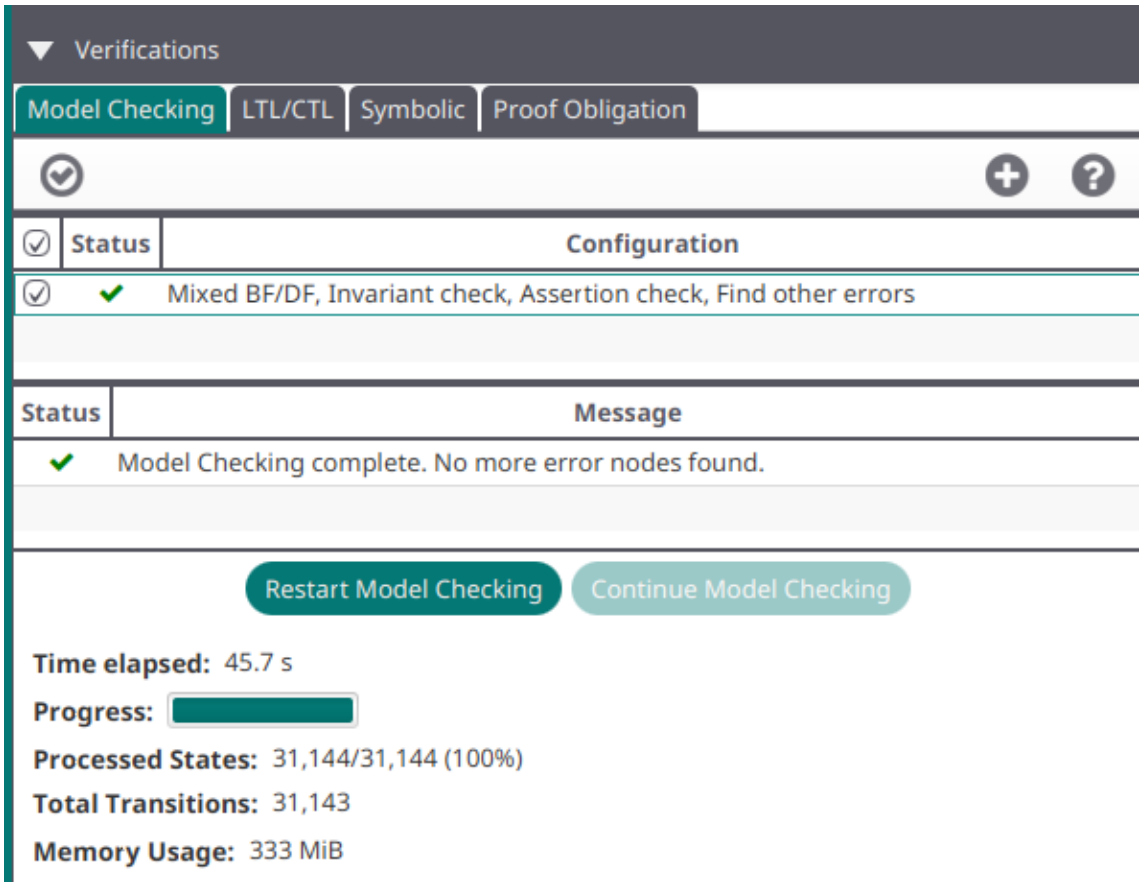

Figure 13: Taking the performance stats from the ProB2-UI *Verifications* interface

9.1 Reducing the state-space size with partial order reduction

To mitigate the state-space explosion problem, we can find parallel paths that contain the same transitions in a different order and lead to the same target state. This technique is called partial order reduction [Cla+99] or POR for short, but in high-level formalisms like B and TLA$^+$ it may not have the desired effects, as shown by Körner and Leuschel [KL23].

To compare the impact of POR we perform benchmarks, putting special attention on

measuring the model checking performance with a specific protocol. We choose the Needham-Schroeder protocol with Lowe's fix (see Section 9.5), processed with the `b` output mode and preset for this, as it is a non-trivial protocol that fulfills both secrecy and authentication properties, so the measured performance corresponds to an exploration of the whole state-space.

We have three different settings we can change.

1. Setting the native ProB POR preference, which may be one of `off`, `ample_sets`, `ample_sets2`.
2. Adding the `phase` variable and additional guards.
3. Adding additional guards during the run phase so that the term unpacking operations are always executed in a fixed order before the protocol execution advances.

The results are shown in Table 11. It is immediately obvious that the POR preference and the manual POR via additional guards do not improve performance at all, the runtime is at least twice as long. The gains achieved by reducing the number of states or transitions seem to be outpaced by the slowdown from the additional guards. Enabling the `phase` variable only has a minor negative impact on performance, but it helps with animation and debugging, so it can stay enabled by default.

POR preference	manual POR: phases	manual POR: execution	States	Transitions	Time
default (`off`)	`false`	`false`	31 071	37 024	43.2 s
default (`off`)	`true`	`false`	31 144	36 793	43.9 s
default (`off`)	`false`	`true`	25 667	26 142	159.4 s
default (`off`)	`true`	`true`	25 740	25 911	160.3 s
`ample_sets`	`false`	`false`	31 071	37 024	89.0 s
`ample_sets2`	`false`	`false`	31 071	37 024	89.2 s

Table 11: Model checking performance for different POR settings using the fixed Needham-Schroeder protocol with the `b` output mode

9.2 Using symmetry reduction to remove equivalent thread initializations

Another way to mitigate the state-space explosion problem is to reduce the amount of states that have different values for variables, that are equivalent under some notion. This

technique is called symmetry reduction [Leu+06; LM10; Cla+99].

In this case special attention has been given to the initialization of threads at the beginning of protocol execution. Our notion of equality for agents is as follows: agents do not share knowledge between protocol runs, so they are functionally identical. It does not matter whether Alice talks with Bob or someone else, and similarly it does not matter what roles Alice and Bob take in their protocol run. The only difference between agents is the Compromised status, which marks agents that had their private keys breached. The Dolev-Yao DSL introduces agents that have this property, named `Compromised1`, ..., `CompromisedN` and the honest agents are named `Alice`, `Bob` and so on. In theory one honest agent and one compromised agent would suffice, but for readability and visualization purposes, there will be as many agents as there are roles in the protocol, so the canonical honest agent-to-role assignment consists of each distinct honest agent assigned to the roles in an arbitrary but fixed order. This result is backed by Comon-Lundh and Cortier [CC04].

A role-to-agent assignment for a simple protocol might look like $\{I \mapsto \text{Alice}, R \mapsto \text{Bob}\}$, which is equivalent to $\{I \mapsto \text{Bob}, R \mapsto \text{Alice}\}$, but distinct from $\{I \mapsto \text{Compromised}, R \mapsto \text{Alice}\}$ and $\{I \mapsto \text{Alice}, R \mapsto \text{Compromised}\}$. Whereas $\{I \mapsto \text{Compromised}, R \mapsto \text{Alice}\}$ is equivalent to $\{I \mapsto \text{Compromised}, R \mapsto \text{Bob}\}$. In addition to that each thread also needs a role which would double the possibility space again.

A naive assignment for this example protocol with two roles and four agents (two honest and two compromised) for each role would create a total of $2 \cdot 4 \cdot 3 = 24$ different assignments. This is equivalent to drawing an ordered subset of size #Roles from #Agents elements without duplicates (no putting back elements between draws). In general, we have for the number of thread initializations #TI:

$$\#\text{TI} = \#\text{Roles} \cdot \frac{\#\text{Agents}!}{(\#\text{Agents} - \#\text{Roles})!} \tag{1}$$

In our case of n roles and n honest and compromised agents each this simplifies to $\#\text{TI}(n) = n \cdot \frac{(2n)!}{(2n-n)!} = \frac{(2n)!}{(n-1)!} \in \mathcal{O}((2n)!)$.

Removing equivalent assignments according to the previous definition leaves us with 4 (Alice with Bob, Compromised with Alice, Alice with Compromised and some Compromised agent with another Compromised agent) different assignments and thus $2 \cdot 4 = 8$ different initializations. The internal order of honest agents and compromised agents respectively does not matter. For each role there are two possibilities, either take the honest or the compromised agent, which leads to the general formula

$$\#\text{TI} = \#\text{Roles} \cdot 2^{\#\text{Roles}} \tag{2}$$

This function is in $\mathcal{O}\left(\#\text{Roles} \cdot 2^{\#\text{Roles}}\right)$.

So the symmetry reduction has improved the asymptotic behavior from factorial to super-exponential, which are distinct classes and thus is a big improvement. Exact values can be seen in Table 12 and Figure 14.

Role count	Thread initializations	Thread initializations (with symmetry reduction)
2	24	8
3	360	24
4	6720	64
5	151 200	160
6	3 991 680	384
7	121 080 960	896
8	4 151 347 200	2048

Table 12: Number of different thread initializations by protocol role count using default agent counts

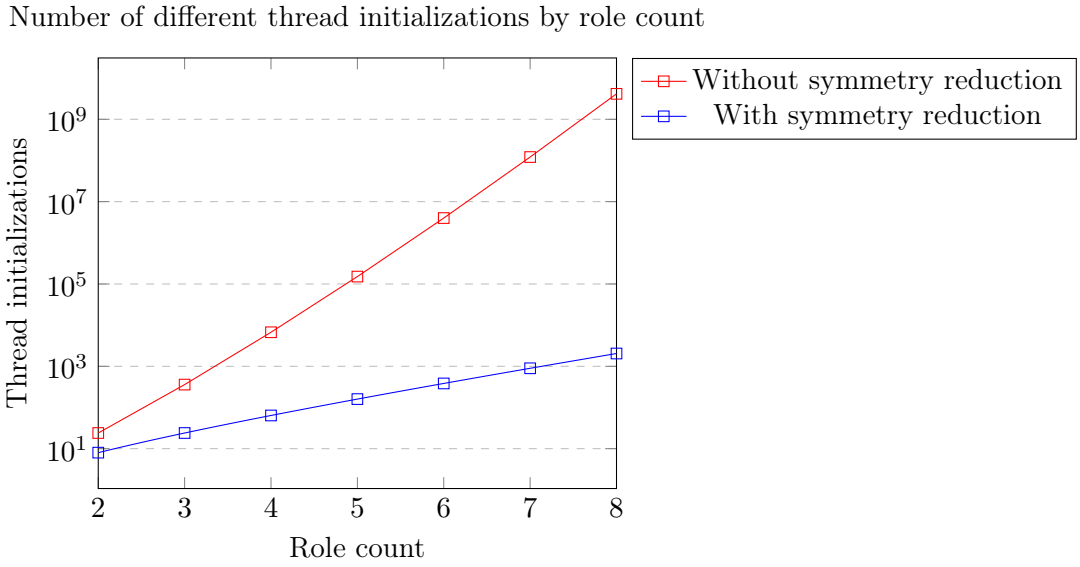

Figure 14: Number of different thread initializations by protocol role count using default agent counts

ProB also has native symmetry reduction, accessed using a preference, which can be set to one of multiple different modes:

1. `off`, default setting, no symmetry reduction is applied
2. `flood`, uses permutation flooding for symmetry reduction [Leu+06]
3. `nauty`, uses the *nauty* graph isomorphism library for symmetry reduction [SL08]
4. `hash`, uses state hashing for symmetry reduction [LM10]

We again perform a benchmark using the Needham-Schroeder protocol with Lowe's fix. The results can be seen in Table 13, and show that symmetry reduction has a negative impact on performance. The `nauty` test did not even terminate, after not progressing for 15 min it was canceled. Thus, symmetry reduction will stay disabled by default.

symmetry preference	States	Transitions	Time
default (`off`)	31 144	36 793	42.9 s
`flood`	31 144	36 793	44.2 s
`nauty`	N/A	N/A	N/A
`hash`	31 144	36 793	44.6 s

Table 13: Model checking performance for different symmetry settings using the fixed Needham-Schroeder protocol with the `b` output mode

9.3 Limitation on the number of threads and agents

When generating a new model in any output format, there will be a hard limit on the number of agents and the number of threads that can be created. By default, there can be as many honest agents as there are communicating agents in the protocol, which is two for the examples and case studies examined in this thesis. In addition to those honest agents there will always be as many compromised agents, whose private key is known to the adversary, as there are honest agents, so there are four agents in total in our examples, as explained by Section 9.2. And there are as many threads allowed as there are honest agents, which amounts to two threads.

Those limits are required to make the state-space finite, as the mathematical model allows the arbitrary creation of threads at any time, with no limits. Explicit state model checking must be able to visit all states to prove the correctness of the given system, which is of course impossible when there are infinite states. But even increasing the limit beyond two leads to state-space explosion as shown in Table 14. When specifying a maximum of three threads, the state-space exploration does not finish in a realistic timeframe.

So it is required that the maximum thread count stays at two to make model checking feasible. Assuming a role count of two, there can be at most two parallel instances of a protocol and the adversary implicitly takes the role of the communication partner in both threads. Thus replay attacks can be tested against. But this property breaks with bigger role counts, as even fully honest protocol runs cannot be simulated anymore. There are simply not enough available threads for each honest participant. Thus, we limit ourselves to protocols with two roles. But limiting the thread count is still an approximation and may lead to missed attacks, as explained by Comon-Lundh and Cortier [CC04], even though limiting the agent count is allowed. The Dolev-Yao DSL limits the possible properties that one can formulate however, and it also limits the messages that can be sent and

received, by not allowing variables to contain anything other than fresh values. With these simplifications the security protocol is not as powerful and should be able to be model checked using this approximation. All tested protocols gave the expected result, but a formal proof of this conjecture is still pending.

Max threads	States	Transitions
0	3	2
1	49	48
2	11 919	14 652
3	more than 3 000 000 (does not finish)	more than 3 600 000 (does not finish)

Table 14: State-space size of fixed Needham-Schroeder XTL model by maximum threads

9.4 Comparison of ProB model checkers

Once again we perform a benchmark using the Needham-Schroeder protocol with Lowe's fix, this time to compare the different model-checkers supported by ProB.

To find the best performing settings for the ProB model checker we first compare the model checking statistics of the `b` output mode with different settings. The settings are *timeout*, *max operations*, *operation reuse* and *state compression*. *Timeout* controls how long the exploration of a state and its outgoing transitions may take, but keeping track of it may hurt the performance. *Max operations* controls how many outgoing transitions from a given state ProB shall explore. If this number is too small the whole state-space will not be explored. *Operation reuse* will cache the projected state-space for each operation and *state compression* will compress states in memory at the cost of performance.

The results are shown in Table 15. Enabling *operation reuse* gives a significant performance increase of 24 % for the partial setting and 26 % for the full setting, but this comes at the cost of a 283 % or 350 % increase of memory usage respectively. Whereas enabling *state compression* increases the runtime by barely 1 % but reduces memory usage by 31 % for both settings. Disabling the timeout leads to a minor improvement in performance of 6 %, while increasing *max operations* has no significant impact. Combining all settings leads to a runtime reduction of 37 % while not significantly affecting the memory usage. Thus, all settings should stay enabled for the ProB model checker to maximize performance.

timeout	*max operations*	*operation reuse*	*state compression*	Time	Memory Usage
default (2.5 s)	default (10)	default (`false`)	default (`false`)	67.9 s	332 MiB
disabled	default (10)	default (`false`)	default (`false`)	63.5 s	332 MiB
default (2.5 s)	1 000 000	default (`false`)	default (`false`)	67.8 s	332 MiB
default (2.5 s)	default (10)	`true`	default (`false`)	51.6 s	1274 MiB
default (2.5 s)	default (10)	`full`	default (`false`)	50.2 s	1495 MiB
default (2.5 s)	default (10)	default (`false`)	`true`	68.5 s	229 MiB
default (2.5 s)	default (10)	default (`false`)	`full`	68.4 s	229 MiB
disabled	1 000 000	`full`	default (`false`)	45.7 s	1495 MiB
disabled	1 000 000	`full`	`full`	42.6 s	334 MiB

Table 15: Model checking performance for different settings using the fixed Needham-Schroeder protocol with the `b` output mode

To run the benchmark, we generate output with all modes and presets and perform a model check with the ProB toolchain under different configurations. ProB itself has built-in support for multiple explicit state model checkers. These are the standard ProB model checker, the TLC model checker built for TLA$^+$ and the LTSmin model checker [KLM18; Kan+15].

The results are shown in Table 16. For this model the TLC model checker is the fastest, even though it has to check the most states with around 30 000, followed by the ProB model checker for the XTL variant of the model. The LTSmin model checker is the slowest and is nearly twice as slow as the ProB model checker for the `b` output mode. Model checking of the B models with the ProB model checker is an order of magnitude slower than checking the XTL model. Using freetypes with the `visualization` preset or using the recursive message terms with the `xtl` mode cuts the size of the state-space nearly to a third, and it also reduces the model checking runtime by 50 % or 95 % respectively. This happens due to the addition of the `all_terms` variable which collects all known terms *in a specific order*, and thus making some states distinct that have the same known terms, just discovered in a different order. This is reminiscent of partial order reduction. Model checking the back-translated TLA$^+$ or Event-B models is infeasible because the runtime is - with ten times the base value - just too high in comparison. Finally, we can conclude that using the TLC model checker gives the best performance, so comparisons with existing

tools will be done with this backend and model checker.

Mode or preset	Model checker	States	Transitions	Time
`b`	ProB	31 144	36 793	42.4 s
`b`	LTSmin	31 143	36 792	76.7 s
`b` with `visualization` preset	ProB	11 714	19 599	21.0 s
`b` with `visualization` preset	LTSmin	11 713	19 598	41.2 s
`b` with `tla` preset	ProB	31 144	36 793	44.0 s
`b` with `tla` preset	TLC	31 142	36 930	0.7 s
`b` with `tla` preset	LTSmin	31 143	36 792	81.0 s
The TLA$^+$ module generated by TLC4B from the `tla` preset	ProB	31 143	36 792	390.1 s
The TLA$^+$ module generated by TLC4B from the `tla` preset	LTSmin	31 143	36 792	393.7 s
`eventb`	ProB	31 144	36 793	287.4 s
`xtl`	ProB	11 919	14 652	1.9 s

Table 16: Model checking performance for the various backends using the fixed Needham-Schroeder protocol

9.5 Case Study: The Needham-Schroeder protocol

The Needham-Schroeder public key protocol [NS78] is used to authenticate two agents which each other by sharing two freshly generated secrets, that can only be read when both parties know the correct private keys. It is used as a part or prerequisite of other protocols, to ensure that two agents are actually talking to each other instead of an impersonator.

After its inception it stood unchallenged for 17 years before Lowe demonstrated a security flaw [Low95] in it using the FDR model checker [Low96]. In this case study we implement both the original and fixed Needham-Schroeder in our Dolev-Yao DSL, in ProVerif, in

Scyther, the Tamarin prover and AVISPA to search for the breach and to compare the results.

The Needham-Schroeder protocol in its initial definition uses a trusted public key server, that is used to provide the public keys of all agents, but in this simplified version it is assumed that all agents know each other's public key. This is also part of the initial state of our Dolev-Yao DSL semantic. In its simplified form the protocol consists of two interleaved exchanges between the two agents: both generate a nonce, encrypt it, send it, and wait for the other party to respond with the same nonce. If both exchanges are successful both parties can be assured that there are no impostors. A sequence diagram of the protocol is shown in Figure 15.

When the initiator, let us call her Alice, sends the opening message containing her identity and a nonce, only the intended recipient, which we call Bob, can read them, because only he is able to decrypt the message using his secret key. With the identity of Alice now known to Bob, he can look up her public key for encryption of the return message, which contains the received nonce and a freshly generated nonce for the second exchange. Only Alice can decrypt this response containing the two nonces and compare the first one with the nonce she generated in the beginning. If it matches, she can deduce that her communication partner is Bob, because only he could have read her first message. The second nonce is sent back to Bob, encrypted with his public key. Similarly, Bob receives this last message, decrypts it, compares its contents with his nonce, and if successful deduces he must really be talking to Alice. Now all agents are sure that they share the same nonces, and they are speaking to each other.

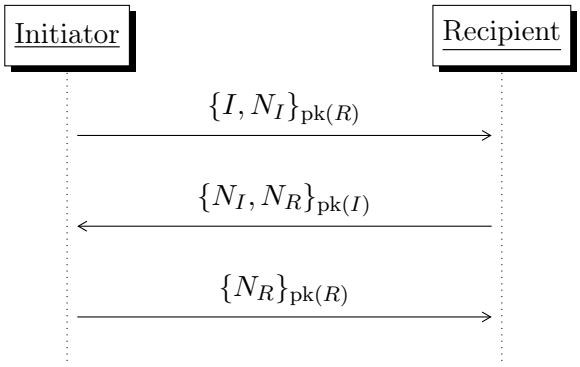

Figure 15: Needham-Schroeder public key protocol, N_I and N_R are freshly generated nonces

But unfortunately this logic is incorrect, as shown in Figure 16. An adversary can reuse nonces by replaying messages. The protocol does not guarantee who generated which nonce and if the nonces are supposed to be used for the current run.

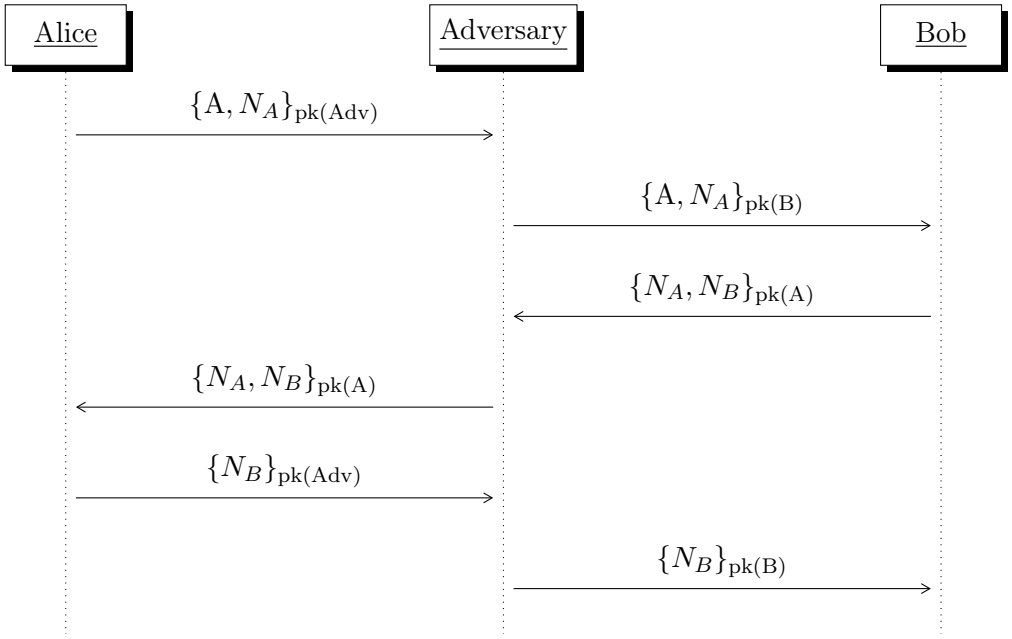

Figure 16: Lowe's replay attack on the Needham-Schroeder public key protocol, N_A and N_B are freshly generated nonces

In the attack Alice is doing a normal run of the protocol with a compromised or adversarial agent, which in turn is running a normal protocol instance with Bob, while *pretending to be Alice*. After the successful run of both protocols, Alice has correctly authenticated with the adversarial agent and Bob wrongly believes he authenticated with Alice.

Anyone can send the first message to Bob, because it is encrypted with his public key. The second message is the crux of the matter, it is supposed to guarantee to Alice that Bob is on the other side, and through the final message ensure Bob of the same. But Bob does not know if his partner intended to communicate with him. The fix proposed by Lowe, as presented in Figure 17, has Bob sending his own identity in the form of his public key. A different fix would change the messages to be doubly encrypted, with the private key of the sender and public key of the intended receiver. This is similar to Lowe's fix, as now the identity of both the sender and receiver are part of the messages.

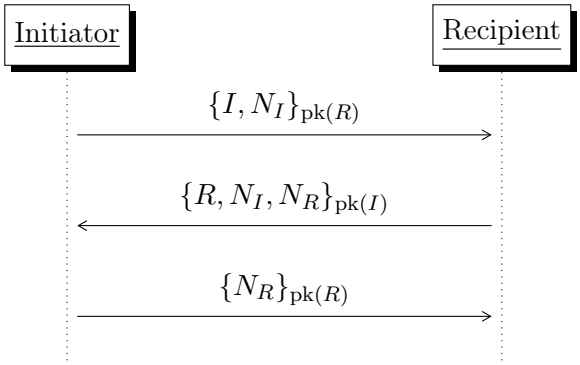

Figure 17: Needham-Schroeder public key protocol with Lowe's fix, N_I and N_R are freshly generated nonces

Next we demonstrate how to implement the Needham-Schroeder protocol in the Dolev-Yao DSL and all tools mentioned in Section 2. The implementations are mostly taken from the official examples for each tool, with slight adjustments to make them more similar and to add some documentation.

9.5.1 *dy2prob*

The Needham-Schroeder protocol and its properties are implemented in the Dolev-Yao DSL as shown in Listing 43. Note that instead of sending the agents' identities, the public keys are sent. All generated machines will find a trace to an invariant violation, and as demonstrated in Table 15, the TLC model checker will find it in less than a second. When using the `visualization` preset a UML sequence can be generated that shows a protocol run leading to the invariant violation, which looks like Figure 18.

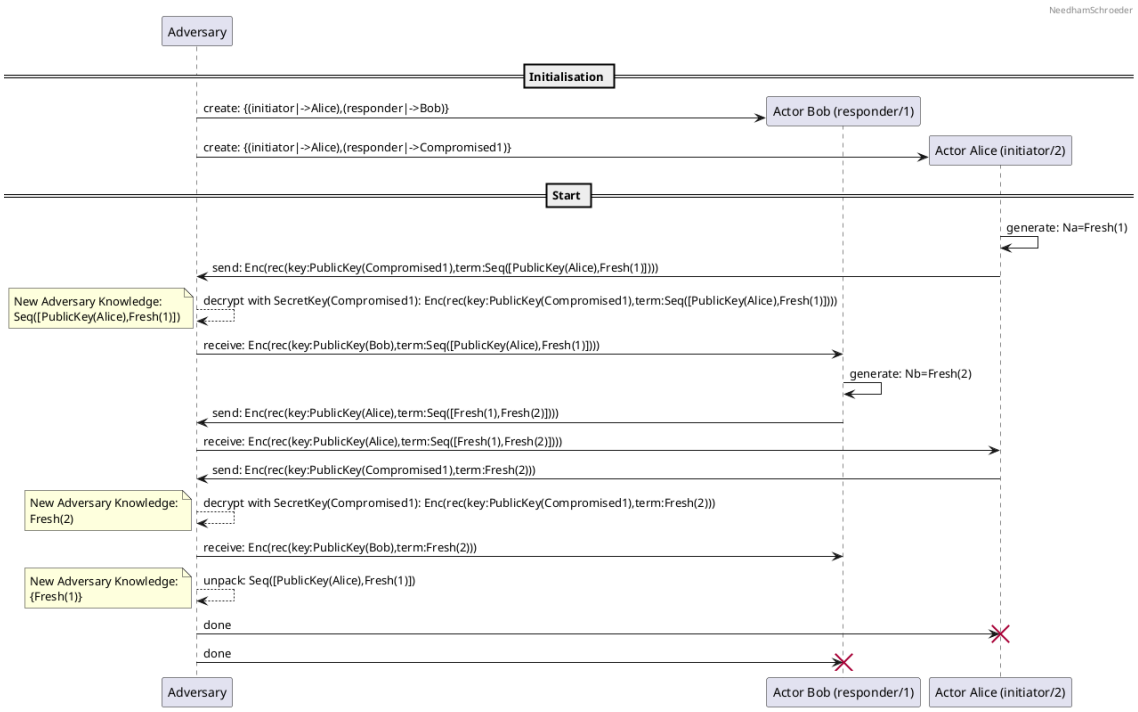

Figure 18: UML sequence chart of the attack on the Needham-Schroeder protocol

After applying Lowe's fix to the protocol the second message changes to include the initiator's public key, as shown in Listing 44.

9.5.2 ProVerif

The implementation of the Needham-Schroeder protocol for ProVerif can be seen in Listing 45. Notably it is very large and contains an implementation of encryption and secrecy. It also has to declare a root process that setups the real processes for Alice and Bob.

Executing the validation takes roughly 0.1 s with ProVerif 2.05 and generates console output explaining the attack. When passing the `-html` command-line switch it generates a HTML page containing a sequence chart (Figure 19) of an attack trace. Although the trace displays the same attack, it is hard to comprehend, because it does not only show the messages transmitted, but also the events used to implement secrecy and the term rewriting rules that define encryption.

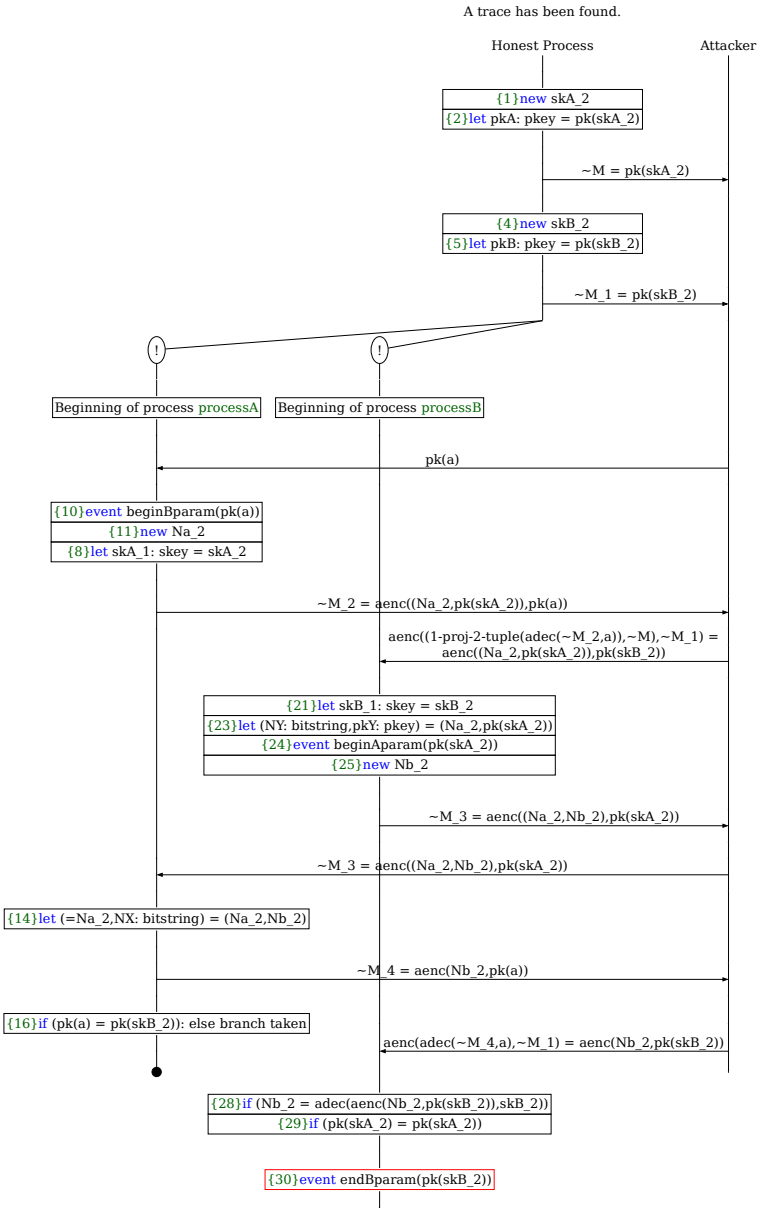

Figure 19: Needham-Schroeder property violation trace found by ProVerif

After applying Lowe's fix, shown in Listing 46, there are no further property violations reported.

9.5.3 Scyther

The Scyther application is a graphical user interfaces which contains an editor and a simple menu to start the verification process. We are using Scyther version 1.1.3.

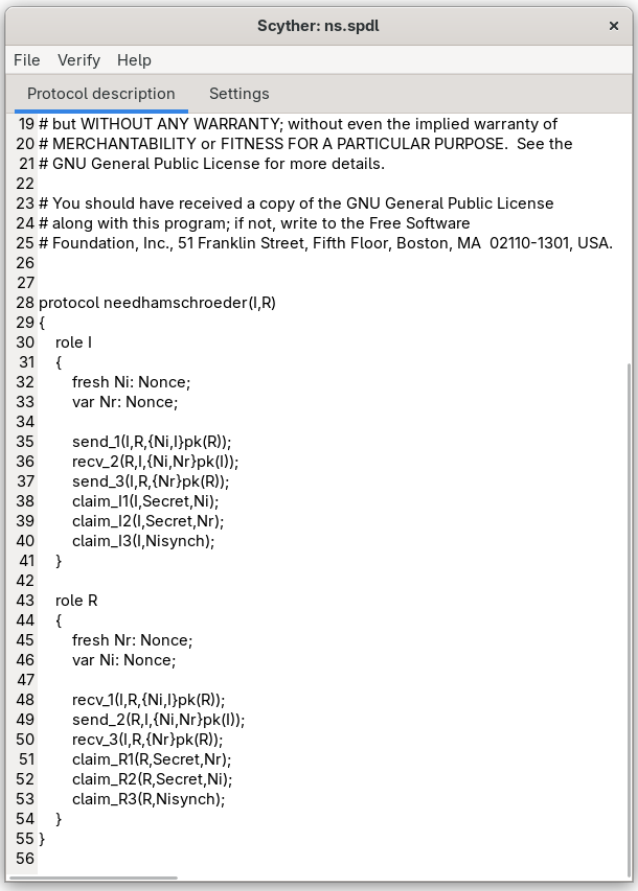

Figure 20: Scyther GUI

The implementation of the Needham-Schroeder protocol closely follows the notation demonstrated by the formal Dolev-Yao model, but one has to declare the generated fresh values and the used variables. It can be seen in Listing 47, while the fixed variant is shown in Listing 48. A trace breaking the stated secrecy properties is found instantly and yields the screen shown in Figure 23, which leads to the actual trace in Figure 21 – in this case demonstrating how the secrecy of the recipient's nonce is violated. As with the other tools, applying Lowe's fix fixes the reported problem and no further violations are reported.

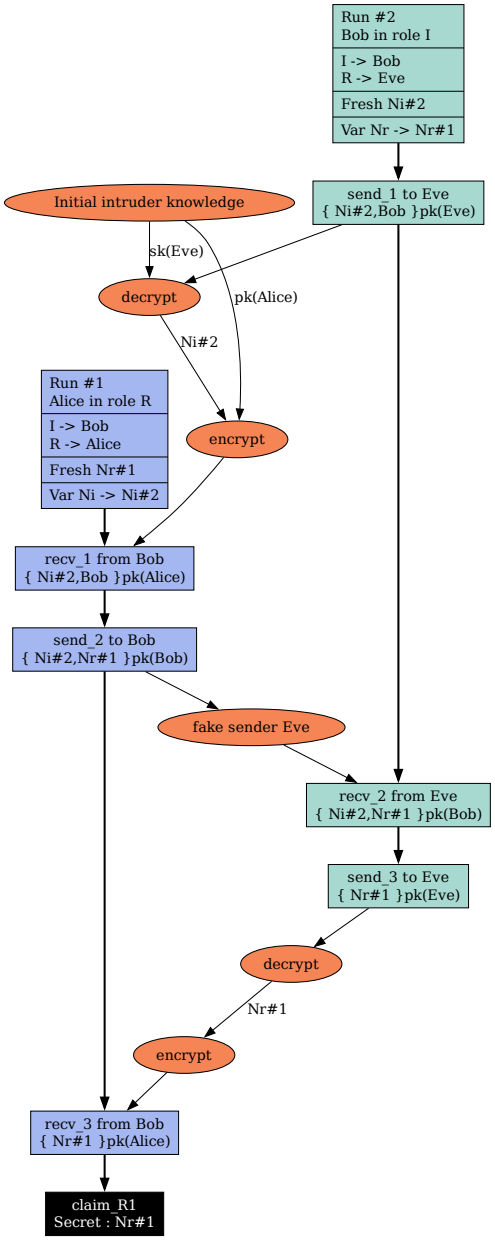

Figure 21: Scyther attack trace

9.5.4 Tamarin

The Tamarin Prover uses the Maude theorem prover as a backend to prove the given security properties. The Needham-Schroeder protocol as a *Security Protocol Theory*, the input format for Tamarin, looks like Listing 49, while the fixed version looks like Listing 50. The implementation is very verbose and uses a different syntax than the Dolev-Yao derived representation of Scyther. Running the Tamarin prover version 1.10.0 through the command-line interface takes around 4 s, while also generating the visualizations for the counter-examples to the protocol's security properties Figures 24 to 26. The output log (Listing 41) states what property has failed verification, but the generated graphs showing the traces are incomprehensible. As expected the fixed protocol reports no property violations.

Listing 41: Needham-Schroeder results with Tamarin

```
1:  ==============================================================================
2:  summary of summaries:
3:
4:  analyzed: ns.spthy
5:
6:    output:           out/ns_analyzed.spthy
7:    processing time: 2.37s
8:
9:    types (all-traces): verified (33 steps)
10:   nonce_secrecy (all-traces): falsified - found trace (16 steps)
11:   injective_agree (all-traces): falsified - found trace (14 steps)
12:   session_key_setup_possible (exists-trace): verified (5 steps)
13:
14: ==============================================================================
```

9.5.5 AVISPA

AVISPA is a collection of multiple different command-line tools for protocol verification with a common front-end. The implementation of the Needham-Schroeder protocol looks like Listing 51, while Lowe's fix takes the form of Listing 52. Note how each role keeps track of its progress in the protocol with a counter and all variables have to be declared before use.

Due to its age and the non-functioning website it was not possible to get all parts of the toolchain to work. These tests were performed with a version of AVISPA from May 2006. The SAT backend did not work at all and the T4ASP backend did not find a trace leading to a property violation. But the OFMC backend succeeded and instantly found a counter-example, however the results are printed as a textual report in the console, as shown in Listing 42. The log contains the attack trace in text form. After fixing the protocol, the

report states the result *SAFE*.

Listing 42: Needham-Schroeder results with AVISPA OFMC

```
% OFMC
% Version of 2006/02/13
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  NSPK.if
GOAL
  secrecy_of_nb
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.00s
  visitedNodes: 8 nodes
  depth: 2 plies
ATTACK TRACE
i -> (a,6): start
(a,6) -> i: {Na(1).a}_ki
i -> (b,3): {Na(1).a}_kb
(b,3) -> i: {Na(1).Nb(2)}_ka
i -> (a,6): {Na(1).Nb(2)}_ka
(a,6) -> i: {Nb(2)}_ki
i -> (i,17): Nb(2)
i -> (i,17): Nb(2)
```

10 Conclusion and outlook

In this work a domain-specific language (DSL) was designed to represent security protocols with the Dolev-Yao formal model. Then a tool was developed on top of *lisb* to automatically translate this representation into classical B, Event-B and XTL, for further processing with the ProB toolchain, which had to be extended to support all required functionality. Using ProB gives the ability to verify the protocol properties using model checking, to manually animate it and to generate UML sequence chart visualizations. Finally, comparisons and benchmarks were performed to find the most performant model checker and preferences supported by ProB, which were the TLC model checker running with multiple workers, and to compare the results with other established automatic protocol verification tools.

It was possible to demonstrate the identification of a known security flaw in the famous Needham-Schroeder protocol. The verification performance was comparable to existing tools when using the TLC model checker or the XTL representation. Using classical B allowed animation and visualization, on an equal or better level than existing tools.

Due to ProB being built around an explicit state model checker, it is susceptible to the state-space explosion problem, which was encountered when scaling the number of roles or threads. Additionally, the state-space has to be finite to allow a complete check, but this was only possible to achieve when limiting the number of concurrent threads executing the protocol, which is an approximation and may lead to missing attacks. To mitigate this, the DSL was kept simple to restrict the complexity of the specifiable protocols and properties. Thus, this approach using explicit state model checking can only show attacks but not prove correctness. But *dy2prob* can be a good tool to help visualize protocol execution and can be used in teaching and other less formal settings.

10.1 Future work

In the future, the Dolev-Yao DSL could be extended to include more functions like *XOR*. This would allow a user to model more security protocols. To make the Dolev-Yao DSL more user-friendly, and the static checks could be extended, which would notify the user of semantic errors, for example if a term could never be decrypted.

Another avenue of improvement would be the ProB model checker itself, which is significantly slower when model checking the generated Dolev-Yao models than the TLC model checker. A deeper investigation of the causes of this might yield possible optimizations, both in ProB and the generated models. However, this would not make model checking a bigger protocol utilizing more than two roles feasible, because of the state-space explosion problem.

To tackle the state-space explosion problem, the automatic generation of a provable Event-B model should be considered, as proving correctness allows for infinite state-spaces. It would

not be a realistic requirement that all proof obligations pass automatically, user intervention is certainly needed. For this to work, improving the Event-B representation of terms is necessary, for example using the Event-B theory plugin, as well as strengthening all invariants and guards. Alternatively, the symbolic model checking native to ProB could be evaluated.

Finally, different output modes or backends could be considered, one prime candidate being b2program [Vu+19; VBL22], which allows generating a native application for model checking a specification. It was not possible to implement a b2program backend in this work because nested guards and nondeterminism are not currently supported. Another backend could be inspired by Lowe's work, as he found the flaw in the Needham-Schroeder protocol using the FDR model checker, which uses CSP (Communicating Sequential Processes) [Hoa85] as an input language. And ProB has an interpreter for CSP as well [Leu01], so this is another backend worth considering.

List of Tables

List of Figures

List of Listings

Appendices

A *dy2prob*

A.1 Source Code

The source code of *dy2prob* can be found at https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/vella-master-code in the `src` directory. A lot of examples protocols can be found in the `examples` directory. All case study files can be found in the `case_study` directory.

A.2 Building and running

Run `make jar` to build a standalone jar in the `target` directory which can be executed with `java -jar dy2prob-0.1.0-SNAPSHOT-standalone.jar`.

Run `make examples` to process all examples to generate B, Event-B and XTL machines into `out_b`, `out_eventb` and `out_xtl` directories respectively.

Run `make test` to execute the test suite, which consists of unit tests that check the preprocessing of the Dolev-Yao DSL and a lot of integration tests that verify the correctness of the translation by comparing the results of model checking with an expected value.

A.3 Trying out the changes in the ProB toolchain

Most changes are already released as version 1.15.0 of the ProB CLI, except for the TLA2BAST and TLC4B changes and ProB2-UI. To obtain a working and up-to-date version of ProB2-UI, it is necessary to check out the specific commit `4876b4d`[10] and to build and run it from source.

[10] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob2_ui/-/commit/4876b4d5ad7e7a75a48b82b419e753804c747b4e

B Case Study

This section contains the specifications of the Needham-Schroeder protocol for each tool used in the case study.

B.1 *dy2prob*

Listing 43: Needham-Schroeder protocol implemented with the Dolev-Yao DSL

```
(protocol
 :NeedhamSchroeder
 (threads
  (:initiator [(generate :Na)
               (send (enc (public-key :responder)
                          (sequence (public-key :initiator) :Na)))
               (receive (enc (public-key :initiator)
                             (sequence :Na :Nb)))
               (send (enc (public-key :responder)
                          :Nb))])
  (:responder [(receive (enc (public-key :responder)
                             (sequence (public-key :initiator) :Na)))
               (generate :Nb)
               (send (enc (public-key :initiator)
                          (sequence :Na :Nb)))
               (receive (enc (public-key :responder)
                             :Nb))]))
 (properties
  (secrecy :initiator/Na :responder/Nb)
  (weak-aliveness :initiator :responder)
  (weak-agreement :initiator :responder)
  (simple-agreement :initiator :responder)))
```

Listing 44: Needham-Schroeder protocol with Lowe's fix implemented with the Dolev-Yao DSL

```
(protocol
 :NeedhamSchroederFixed
 (threads
  (:initiator [(generate :Na)
               (send (enc (public-key :responder)
                          (sequence (public-key :initiator) :Na)))
               (receive (enc (public-key :initiator)
                             (sequence (public-key :responder) :Na :Nb)))
               (send (enc (public-key :responder)
                          :Nb))])
  (:responder [(receive (enc (public-key :responder)
                         (sequence (public-key :initiator) :Na)))
               (generate :Nb)
```

```
14:                 (send (enc (public-key :initiator)
15:                            (sequence (public-key :responder) :Na :Nb)))
16:                 (receive (enc (public-key :responder)
17:                               :Nb))]))
18:   (properties
19:    (secrecy :initiator/Na :responder/Nb)
20:    (weak-aliveness :initiator :responder)
21:    (weak-agreement :initiator :responder)
22:    (simple-agreement :initiator :responder)))
```

B.2 ProVerif

Listing 45: Needham-Schroeder in ProVerif

```
1:  (************************************************************
2:   *                                                          *
3:   *  Cryptographic protocol verifier                         *
4:   *                                                          *
5:   *  Bruno Blanchet, Vincent Cheval, and Marc Sylvestre      *
6:   *                                                          *
7:   *  Copyright (C) INRIA, CNRS 2000-2023                     *
8:   *                                                          *
9:   ************************************************************)
10:
11:  (*
12:
13:      This program is free software; you can redistribute it and/or
            modify
14:      it under the terms of the GNU General Public License as
            published by
15:      the Free Software Foundation; either version 2 of the License,
            or
16:      (at your option) any later version.
17:
18:      This program is distributed in the hope that it will be useful,
19:      but WITHOUT ANY WARRANTY; without even the implied warranty of
20:      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
21:      GNU General Public License for more details (in file LICENSE).
22:
23:      You should have received a copy of the GNU General Public
            License along
24:      with this program; if not, write to the Free Software
            Foundation, Inc.,
25:      51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
26:
27:  *)
28:
29:  (*
```

```
30:
31: Needham-Schroeder public key protocol
32:
33: As distributed in the documentation of the ProVerif tool.
34: With added comments.
35:
36: Message 1: A -> B : aenc((Na, pk(skA)), pk(skB))
37: Message 2: B -> A : aenc((Na, Nb), pk(skA))
38: Message 3: A -> B : aenc(Nb, pk(skB))
39:
40: *)
41:
42: set traceDisplay = long.
43:
44: (* free=global variable, known to attacker *)
45: free c: channel.
46:
47: (* Public key encryption *)
48: type pkey.
49: type skey.
50:
51: fun pk(skey): pkey.
52: fun aenc(bitstring, pkey): bitstring.
53: reduc forall x: bitstring, y: skey; adec(aenc(x, pk(y)), y) = x.
54:
55: (* Signatures *)
56: type spkey.
57: type sskey.
58:
59: fun spk(sskey): spkey.
60: fun sign(bitstring, sskey): bitstring.
61: reduc forall x: bitstring, y: sskey; getmess(sign(x, y)) = x.
62: reduc forall x: bitstring, y: sskey; checksign(sign(x, y), spk(y))
       = x.
63:
64: (* Shared key encryption *)
65: fun senc(bitstring, bitstring): bitstring.
66: reduc forall x: bitstring, y: bitstring; sdec(senc(x, y), y) = x.
67:
68: (* Authentication queries *)
69:
70: (* A's intention to start protocol with pkey *)
71: event beginBparam(pkey).
72: (* B's belief they have talked to A *)
73: event endBparam(pkey).
74: (* B's belief they started the protocol with pkey *)
75: event beginAparam(pkey).
76: (* A's belief they have talked to B *)
77: event endAparam(pkey).
```

```
78:
79: query x: pkey; inj-event(endBparam(x)) ==>
       inj-event(beginBparam(x)).
80: query x: pkey; inj-event(endAparam(x)) ==>
       inj-event(beginAparam(x)).
81:
82: (* Secrecy queries *)
83: free secretANa, secretANb, secretBNa, secretBNb: bitstring
       [private].
84:
85: query
86:     attacker(secretANa);
87:     attacker(secretANb);
88:     attacker(secretBNa);
89:     attacker(secretBNb).
90:
91: (* A *)
92: let processA(pkB: pkey, skA: skey) =
93:     (* let the environment decide with whom to start the protocol
           *)
94:     in(c, pkX: pkey);
95:
96:     (* fire event for beginning of authentication *)
97:     event beginBparam(pkX);
98:
99:     (* generate nonce *)
100:    new Na: bitstring;
101:
102:    (* Message 1: send nonce and identity *)
103:    out(c, aenc((Na, pk(skA)), pkX));
104:
105:    (* wait for Message 2 *)
106:    in(c, m: bitstring);
107:
108:    (* decode it and check for equality with own nonce *)
109:    let (=Na, NX: bitstring) = adec(m, skA) in
110:    (* decode it and check for equality with own nonce and pk *)
111:    (*let (=Na, NX: bitstring, =pkX) = adec(m, skA) in*)
112:
113:    (* Message 3: send their nonce back *)
114:    out(c, aenc(NX, pkX));
115:
116:    (* fire event for end of authentication - but only if it
           really was B *)
117:    if pkX = pkB then
118:    event endAparam(pk(skA));
119:
120:    (*
121:    send 'secretANa' & 'secretANb', encoded with the nonces as key
```

```
122:     if the attacker knows the nonces they can decode the secrets
123:     *)
124:     out(c, senc(secretANa, Na));
125:     out(c, senc(secretANb, NX)).
126:
127: (* B *)
128: let processB(pkA: pkey, skB: skey) =
129:     (* wait for A to send Message 1 *)
130:     in(c, m: bitstring);
131:
132:     (* decode it into nonce and identity *)
133:     let (NY: bitstring, pkY: pkey) = adec(m, skB) in
134:
135:     (* fire event for beginning of authentication *)
136:     event beginAparam(pkY);
137:
138:     (* generate own nonce *)
139:     new Nb: bitstring;
140:
141:     (* Message 2: send both nonces *)
142:     out(c, aenc((NY, Nb), pkY));
143:     (* Fixed Message 2: send own identity as well *)
144:     (*out(c, aenc((NY, Nb, pk(skB)), pkY));*)
145:
146:     (* wait for Message 3 *)
147:     in(c, m3: bitstring);
148:
149:     (* fire event for end of authentication - but only if it
           really was A *)
150:     if Nb = adec(m3, skB) then
151:     if pkY = pkA then
152:     event endBparam(pk(skB));
153:
154:     (*
155:     send 'secretBNa' & 'secretBNb', encoded with the nonces as key
156:     if the attacker knows the nonces they can decode the secrets
157:     *)
158:     out(c, senc(secretBNa, NY));
159:     out(c, senc(secretBNb, Nb)).
160:
161: (* Main *)
162: process
163:     (* create new secret key *)
164:     new skA: skey;
165:     (* construct public key *)
166:     let pkA = pk(skA) in
167:     (* send public key into channel, so attacker knows it *)
168:     out(c, pkA);
169:
```

```
170:      (* same for B *)
171:      new skB: skey; let pkB = pk(skB) in out(c, pkB);
172:
173:      (* start unbounded number of instances *)
174:      ( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

ProVerif results

Process 0 (that is, the initial process)

- Process 1 (that is, process 0, with let moved downwards)
  - Query inj-event(endBparam(x)) ==> inj-event(beginBparam(x)) in process 1.
    Lemmas
    Clauses
    Completed clauses
    - goal reachable: b-event(@p_act(@occ13_1,aenc((Na_1,Nb_1),pk(skA[])))) && b-inj-event(beginBparam(pk(y)),@occ10_1) && attacker(y) -> inj-event(endBparam(pk(skB[])),@occ30_1)
      Abbreviations:
      - Na_1 = Na[pkX = pk(y),!1 = @sid]
      - Nb_1 = Nb[m_1 = aenc((Na_1,pk(skA[])),pk(skB[])),!1 = @sid_1]
      - @occ30_1 = @occ30[m3 = aenc(Nb_1,pk(skB[])),m_1 = aenc((Na_1,pk(skA[])),pk(skB[])),!1 = @sid_1]
      - @occ13_1 = @occ13[!1 = @sid]
      - @occ10_1 = @occ10[pkX = pk(y),!1 = @sid]

      Derivation Explained derivation
      Unify derivation
      Trace
      Trace graph

    RESULT inj-event(endBparam(x)) ==> inj-event(beginBparam(x)) is false.
    RESULT (even event(endBparam(x)) ==> event(beginBparam(x)) is false.)
  - Query inj-event(endAparam(x)) ==> inj-event(beginAparam(x)) in process 1.
    Lemmas
    Clauses
    Completed clauses
    - goal reachable: b-inj-event(beginAparam(pk(skA[])),@occ24_1) -> inj-event(endAparam(pk(skA[])),@occ17_1)
      The hypothesis occurs strictly before the conclusion.
      Abbreviations:
      - Na_1 = Na[pkX = pk(skB[]),!1 = @sid]
      - Nb_1 = Nb[m_1 = aenc((Na_1,pk(skA[])),pk(skB[])),!1 = @sid_1]
      - @occ17_1 = @occ17[m = aenc((Na_1,Nb_1),pk(skA[])),pkX = pk(skB[]),!1 = @sid]
      - @occ24_1 = @occ24[m_1 = aenc((Na_1,pk(skA[])),pk(skB[])),!1 = @sid_1]

    RESULT inj-event(endAparam(x)) ==> inj-event(beginAparam(x)) is true.
  - Query
    - not attacker(secretANa[])
    - not attacker(secretANb[])
    - not attacker(secretBNa[])
    - not attacker(secretBNb[])

    in process 1.
    Clauses
    Completed clauses
    - Query not attacker(secretANa[])
      RESULT not attacker(secretANa[]) is true.
    - Query not attacker(secretANb[])
      RESULT not attacker(secretANb[]) is true.
    - Query not attacker(secretBNa[])
      - goal reachable: attacker(secretBNa[])
        Derivation Explained derivation
        Unify derivation
        Trace
        Trace graph

      RESULT not attacker(secretBNa[]) is false.
    - Query not attacker(secretBNb[])
      - goal reachable: attacker(secretBNb[])
        Derivation Explained derivation
        Unify derivation
        Trace
        Trace graph

      RESULT not attacker(secretBNb[]) is false.

Verification summary:

- Query inj-event(endBparam(x)) ==> inj-event(beginBparam(x)) is false.
- Query inj-event(endAparam(x)) ==> inj-event(beginAparam(x)) is true.
- Query not attacker(secretANa[]) is true.
- Query not attacker(secretANb[]) is true.
- Query not attacker(secretBNa[]) is false.
- Query not attacker(secretBNb[]) is false.

Figure 22: Needham-Schroeder property violation found by ProVerif

Listing 46: Needham-Schroeder with Lowe's fix in ProVerif

```
1:  (*
2:
3:  Needham-Schroeder public key protocol
4:
5:  Message 1: A -> B : aenc((Na, pk(skA)), pk(skB))
6:  Message 2: B -> A : aenc((Na, Nb), pk(skA))
```

```
 7: Message 3: A -> B : aenc(Nb, pk(skB))
 8:
 9: *)
10:
11: set traceDisplay = long.
12:
13: (* free=global variable, known to attacker *)
14: free c: channel.
15:
16: (* Public key encryption *)
17: type pkey.
18: type skey.
19:
20: fun pk(skey): pkey.
21: fun aenc(bitstring, pkey): bitstring.
22: reduc forall x: bitstring, y: skey; adec(aenc(x, pk(y)), y) = x.
23:
24: (* Signatures *)
25: type spkey.
26: type sskey.
27:
28: fun spk(sskey): spkey.
29: fun sign(bitstring, sskey): bitstring.
30: reduc forall x: bitstring, y: sskey; getmess(sign(x, y)) = x.
31: reduc forall x: bitstring, y: sskey; checksign(sign(x, y), spk(y))
       = x.
32:
33: (* Shared key encryption *)
34: fun senc(bitstring, bitstring): bitstring.
35: reduc forall x: bitstring, y: bitstring; sdec(senc(x, y), y) = x.
36:
37: (* Authentication queries *)
38:
39: (* A's intention to start protocol with pkey *)
40: event beginBparam(pkey).
41: (* B's belief they have talked to A *)
42: event endBparam(pkey).
43: (* B's belief they started the protocol with pkey *)
44: event beginAparam(pkey).
45: (* A's belief they have talked to B *)
46: event endAparam(pkey).
47:
48: query x: pkey; inj-event(endBparam(x)) ==>
       inj-event(beginBparam(x)).
49: query x: pkey; inj-event(endAparam(x)) ==>
       inj-event(beginAparam(x)).
50:
51: (* Secrecy queries *)
52: free secretANa, secretANb, secretBNa, secretBNb: bitstring
```

```
    [private].

query
    attacker(secretANa);
    attacker(secretANb);
    attacker(secretBNa);
    attacker(secretBNb).

(* A *)
let processA(pkB: pkey, skA: skey) =
    (* let the environment decide with whom to start the protocol
       *)
    in(c, pkX: pkey);

    (* fire event for beginning of authentication *)
    event beginBparam(pkX);

    (* generate nonce *)
    new Na: bitstring;

    (* Message 1: send nonce and identity *)
    out(c, aenc((Na, pk(skA)), pkX));

    (* wait for Message 2 *)
    in(c, m: bitstring);

    (* decode it and check for equality with own nonce *)
    (*let (=Na, NX: bitstring) = adec(m, skA) in*)
    (* decode it and check for equality with own nonce and pk *)
    let (=Na, NX: bitstring, =pkX) = adec(m, skA) in

    (* Message 3: send their nonce back *)
    out(c, aenc(NX, pkX));

    (* fire event for end of authentication - but only if it
       really was B *)
    if pkX = pkB then
    event endAparam(pk(skA));

    (*
    send 'secretANa' & 'secretANb', encoded with the nonces as key
    if the attacker knows the nonces they can decode the secrets
    *)
    out(c, senc(secretANa, Na));
    out(c, senc(secretANb, NX)).

(* B *)
let processB(pkA: pkey, skB: skey) =
    (* wait for A to send Message 1 *)
```

```
99:     in(c, m: bitstring);
100:
101:    (* decode it into nonce and identity *)
102:    let (NY: bitstring, pkY: pkey) = adec(m, skB) in
103:
104:    (* fire event for beginning of authentication *)
105:    event beginAparam(pkY);
106:
107:    (* generate own nonce *)
108:    new Nb: bitstring;
109:
110:    (* Message 2: send both nonces *)
111:    (*out(c, aenc((NY, Nb), pkY));*)
112:    (* Fixed Message 2: send own identity as well *)
113:    out(c, aenc((NY, Nb, pk(skB)), pkY));
114:
115:    (* wait for Message 3 *)
116:    in(c, m3: bitstring);
117:
118:    (* fire event for end of authentication - but only if it
           really was A *)
119:    if Nb = adec(m3, skB) then
120:    if pkY = pkA then
121:    event endBparam(pk(skB));
122:
123:    (*
124:    send 'secretBNa' & 'secretBNb', encoded with the nonces as key
125:    if the attacker knows the nonces they can decode the secrets
126:    *)
127:    out(c, senc(secretBNa, NY));
128:    out(c, senc(secretBNb, Nb)).
129:
130: (* Main *)
131: process
132:    (* create new secret key *)
133:    new skA: skey;
134:    (* construct public key *)
135:    let pkA = pk(skA) in
136:    (* send public key into channel, so attacker knows it *)
137:    out(c, pkA);
138:
139:    (* same for B *)
140:    new skB: skey; let pkB = pk(skB) in out(c, pkB);
141:
142:    (* start unbounded number of instances *)
143:    ( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

B.3 Scyther

Listing 47: Needham-Schroeder in Scyther

```
1:  # Needham Schroeder Public Key
2:  #
3:  # Modelled after the description in the SPORE library
4:  # http://www.lsv.ens-cachan.fr/spore/nspk.html
5:
6:
7:  # Taken from the Scyther Sources
8:  # Removed the authentication server
9:
10: # Scyther : An automatic verifier for security protocols.
11: # Copyright (C) 2007-2020 Cas Cremers
12:
13: # This program is free software; you can redistribute it and/or
14: # modify it under the terms of the GNU General Public License
15: # as published by the Free Software Foundation; either version 2
16: # of the License, or (at your option) any later version.
17:
18: # This program is distributed in the hope that it will be useful,
19: # but WITHOUT ANY WARRANTY; without even the implied warranty of
20: # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
21: # GNU General Public License for more details.
22:
23: # You should have received a copy of the GNU General Public License
24: # along with this program; if not, write to the Free Software
25: # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
       02110-1301, USA.
26:
27:
28: protocol needhamschroeder(I,R)
29: {
30:     role I
31:     {
32:         fresh Ni: Nonce;
33:         var Nr: Nonce;
34:
35:         send_1(I,R,{Ni,I}pk(R));
36:         recv_2(R,I,{Ni,Nr}pk(I));
37:         send_3(I,R,{Nr}pk(R));
38:         claim_I1(I,Secret,Ni);
39:         claim_I2(I,Secret,Nr);
40:         claim_I3(I,Nisynch);
41:     }
42:
43:     role R
44:     {
45:         fresh Nr: Nonce;
```

```
46:         var Ni: Nonce;
47:
48:         recv_1(I,R,{Ni,I}pk(R));
49:         send_2(R,I,{Ni,Nr}pk(I));
50:         recv_3(I,R,{Nr}pk(R));
51:         claim_R1(R,Secret,Nr);
52:         claim_R2(R,Secret,Ni);
53:         claim_R3(R,Nisynch);
54:     }
55: }
```

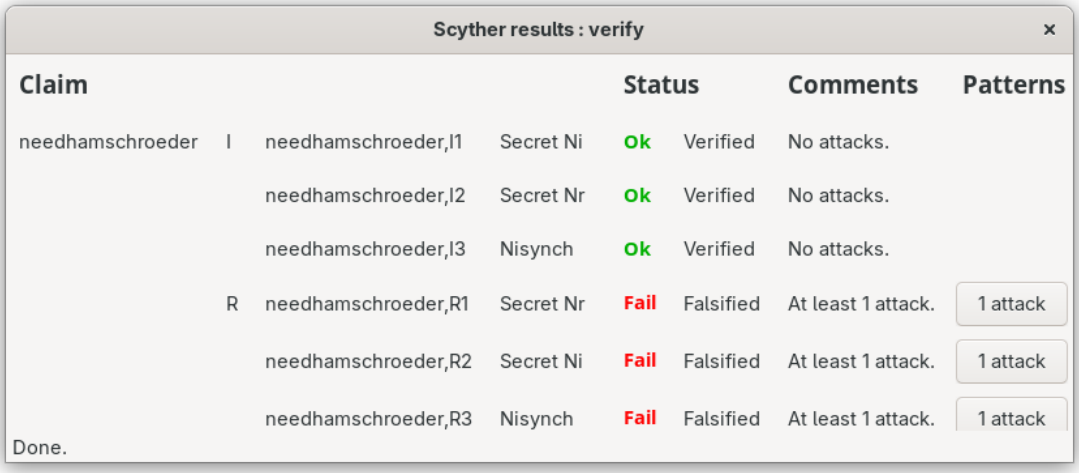

Figure 23: Needham-Schroeder property violation found by Scyther

Listing 48: Needham-Schroeder with Lowe's fix in Scyther

```
1:  # Needham Schroeder Public Key with Lowe's fix
2:  #
3:  # Modelled after the description in the SPORE library
4:  # http://www.lsv.ens-cachan.fr/spore/nspk.html
5:
6:
7:  # Taken from the Scyther Sources
8:  # Removed the authentication server
9:
10: # Scyther : An automatic verifier for security protocols.
11: # Copyright (C) 2007-2020 Cas Cremers
12:
13: # This program is free software; you can redistribute it and/or
14: # modify it under the terms of the GNU General Public License
15: # as published by the Free Software Foundation; either version 2
16: # of the License, or (at your option) any later version.
```

```
17:
18: # This program is distributed in the hope that it will be useful,
19: # but WITHOUT ANY WARRANTY; without even the implied warranty of
20: # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
21: # GNU General Public License for more details.
22:
23: # You should have received a copy of the GNU General Public License
24: # along with this program; if not, write to the Free Software
25: # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
       02110-1301, USA.
26:
27:
28: protocol needhamschroederfixed(I,R)
29: {
30:     role I
31:     {
32:         fresh Ni: Nonce;
33:         var Nr: Nonce;
34:
35:         send_1(I,R,{Ni,I}pk(R));
36:         # recv_2(R,I,{Ni,Nr}pk(I));
37:         recv_2(R,I,{Ni,Nr,R}pk(I));
38:         send_3(I,R,{Nr}pk(R));
39:         claim_I1(I,Secret,Ni);
40:         claim_I2(I,Secret,Nr);
41:         claim_I3(I,Nisynch);
42:     }
43:
44:     role R
45:     {
46:         fresh Nr: Nonce;
47:         var Ni: Nonce;
48:
49:         recv_1(I,R,{Ni,I}pk(R));
50:         # send_2(R,I,{Ni,Nr}pk(I));
51:         send_2(R,I,{Ni,Nr,R}pk(I));
52:         recv_3(I,R,{Nr}pk(R));
53:         claim_R1(R,Secret,Nr);
54:         claim_R2(R,Secret,Ni);
55:         claim_R3(R,Nisynch);
56:     }
57: }
```

B.4 Tamarin

Listing 49: Needham-Schroeder in Tamarin

```
1: theory NSPK3
```

```
2:  begin
3:
4:  builtins: asymmetric-encryption
5:
6:  /*
7:     Protocol:     The classic three message version of the
8:                   flawed Needham-Schroeder Public Key Protocol
9:     Modeler:      Simon Meier
10:    Date:         September 2012
11:
12:    Source:       Gavin Lowe. Breaking and fixing the
          Needham-Schroeder
13:                  public-key protocol using FDR. In Tiziana Margaria
                        and
14:                  Bernhard Steffen, editors, TACAS, volume 1055 of
                        Lecture Notes
15:                  in Computer Science, pages 147-166.  Springer,
                        1996.
16:
17:    Status:       working
18:
19:    Note that we are using explicit global constants for discerning
          the
20:    different encryption instead of the implicit sources.
21:  */
22:
23:
24: // Public key infrastructure
25: rule Register_pk:
26:   [ Fr(~ltkA) ]
27:   -->
28:   [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]
29:
30: rule Reveal_ltk:
31:   [ !Ltk(A, ltkA) ] --[ RevLtk(A)    ]-> [ Out(ltkA) ]
32:
33:
34: /* We formalize the following protocol
35:
36:   protocol NSPK3 {
37:     1. I -> R: {'1',ni,I}pk(R)
38:     2. I <- R: {'2',ni,nr}pk(I)
39:     3. I -> R: {'3',nr}pk(R)
40:   }
41: */
42:
43: rule I_1:
44:   let m1 = aenc{'1', ~ni, $I}pkR
45:   in
```

```
46:       [ Fr(~ni)
47:       , !Pk($R, pkR)
48:       ]
49:     --[ OUT_I_1(m1)
50:       ]->
51:       [ Out( m1 )
52:       , St_I_1($I, $R, ~ni)
53:       ]
54:
55:   rule R_1:
56:     let m1 = aenc{'1', ni, I}pk(ltkR)
57:         m2 = aenc{'2', ni, ~nr}pkI
58:     in
59:       [ !Ltk($R, ltkR)
60:       , In( m1 )
61:       , !Pk(I, pkI)
62:       , Fr(~nr)
63:       ]
64:     --[ IN_R_1_ni( ni, m1 )
65:       , OUT_R_1( m2 )
66:       , Running(I, $R, <'init',ni,~nr>)
67:       ]->
68:       [ Out( m2 )
69:       , St_R_1($R, I, ni, ~nr)
70:       ]
71:
72:   rule I_2:
73:     let m2 = aenc{'2', ni, nr}pk(ltkI)
74:         m3 = aenc{'3', nr}pkR
75:     in
76:       [ St_I_1(I, R, ni)
77:       , !Ltk(I, ltkI)
78:       , In( m2 )
79:       , !Pk(R, pkR)
80:       ]
81:     --[ IN_I_2_nr( nr, m2)
82:       , Commit (I, R, <'init',ni,nr>)  // need to log identities
            explicitely to
83:       , Running(R, I, <'resp',ni,nr>)  // specify that they must not
           be
84:                                        // compromised in the
                                              property.
85:       ]->
86:       [ Out( m3 )
87:       , Secret(I,R,nr)
88:       , Secret(I,R,ni)
89:       ]
90:
91:   rule R_2:
```

```
92:      [ St_R_1(R, I, ni, nr)
93:      , !Ltk(R, ltkR)
94:      , In( aenc{'3', nr}pk(ltkR) )
95:      ]
96:    --[ Commit (R, I, <'resp',ni,nr>)
97:      ]->
98:      [ Secret(R,I,nr)
99:      , Secret(R,I,ni)
100:     ]
101:
102: /* TODO: Also model session-key reveals and adapt security
        properties. */
103: rule Secrecy_claim:
104:   [ Secret(A, B, m) ] --[ Secret(A, B, m) ]-> []
105:
106:
107:
108: /* Note that we are using an untyped protocol model. For proofs,
        we therefore
109: require a protocol specific type invariant for proof construction.
        In
110: principle, such an invariant is not required for attack search,
        but does help
111: a lot.
112:
113: See 'NSLPK3.spthy' for a detailed explanation of the construction
        of this
114: invariant.
115: */
116: lemma types [sources]:
117:   " (All ni m1 #i.
118:        IN_R_1_ni( ni, m1) @ i
119:        ==>
120:        ( (Ex #j. KU(ni) @ j & j < i)
121:        | (Ex #j. OUT_I_1( m1 ) @ j)
122:        )
123:     )
124:   & (All nr m2 #i.
125:        IN_I_2_nr( nr, m2) @ i
126:        ==>
127:        ( (Ex #j. KU(nr) @ j & j < i)
128:        | (Ex #j. OUT_R_1( m2 ) @ j)
129:        )
130:     )
131:   "
132:
133: // Nonce secrecy from the perspective of both the initiator and
        the responder.
134: lemma nonce_secrecy:
```

```
135:   " /* It cannot be that */
136:     not(
137:         Ex A B s #i.
138:           /* somebody claims to have setup a shared secret, */
139:           Secret(A, B, s) @ i
140:           /* but the adversary knows it */
141:         & (Ex #j. K(s) @ j)
142:           /* without having performed a long-term key reveal. */
143:         & not (Ex #r. RevLtk(A) @ r)
144:         & not (Ex #r. RevLtk(B) @ r)
145:        )"
146:
147: // Injective agreement from the perspective of both the initiator
       and the responder.
148: lemma injective_agree:
149:   " /* Whenever somebody commits to running a session, then*/
150:     All actor peer params #i.
151:         Commit(actor, peer, params) @ i
152:       ==>
153:         /* there is somebody running a session with the same
                parameters */
154:           (Ex #j. Running(actor, peer, params) @ j & j < i
155:             /* and there is no other commit on the same parameters
                   */
156:             & not(Ex actor2 peer2 #i2.
157:                     Commit(actor2, peer2, params) @ i2 & not(#i =
                           #i2)
158:                  )
159:           )
160:         /* or the adversary perform a long-term key reveal on
                actor or peer */
161:         | (Ex #r. RevLtk(actor) @ r)
162:         | (Ex #r. RevLtk(peer)  @ r)
163:   "
164:
165: // Consistency check: ensure that secrets can be shared between
       honest agents.
166: lemma session_key_setup_possible:
167:   exists-trace
168:   " /* It is possible that */
169:     Ex A B s #i.
170:       /* somebody claims to have setup a shared secret, */
171:       Secret(A, B, s) @ i
172:       /* without the adversary having performed a long-term key
              reveal. */
173:     & not (Ex #r. RevLtk(A) @ r)
174:     & not (Ex #r. RevLtk(B) @ r)
175:   "
176:
```

```
177: end
```

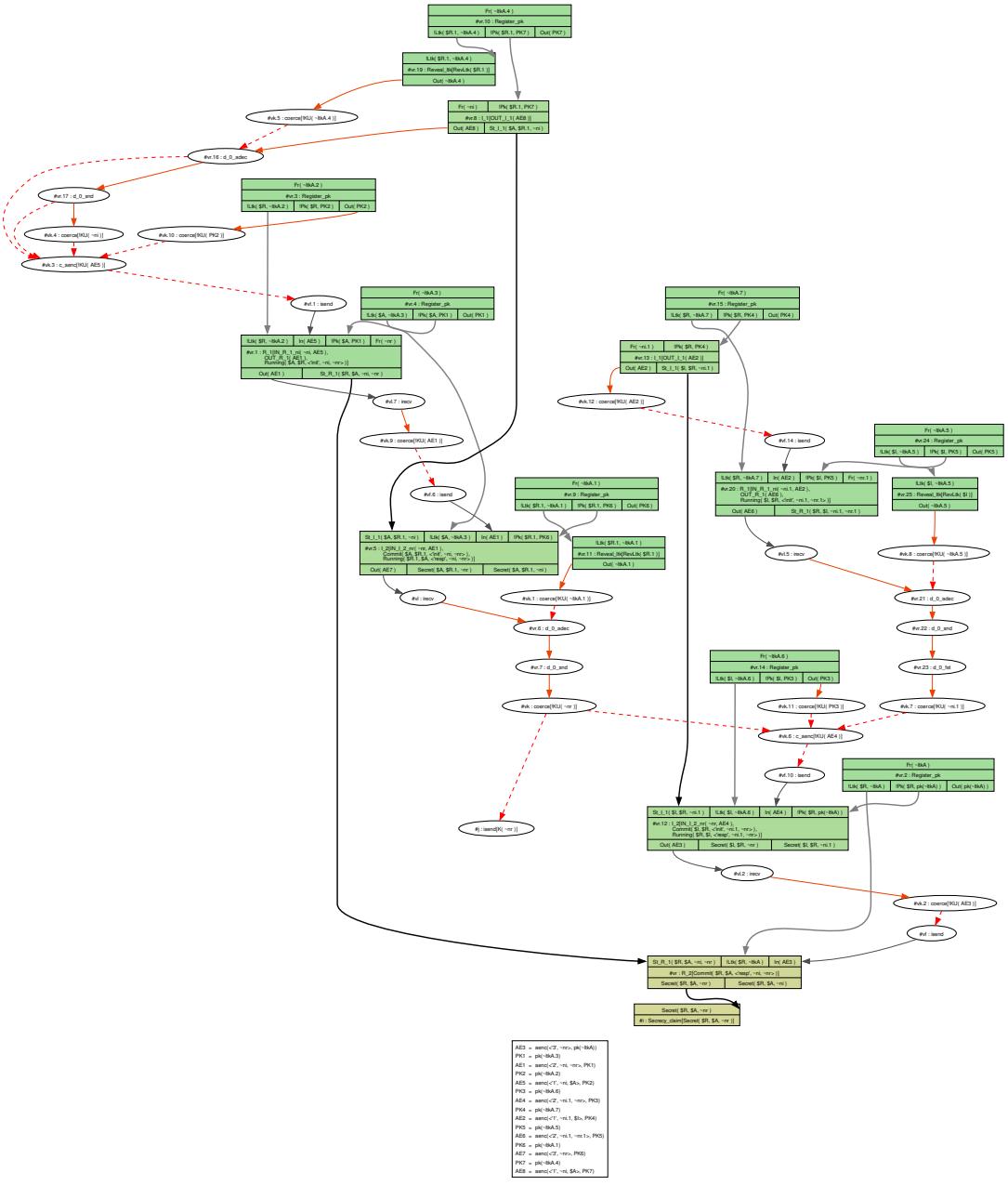

Figure 24: First Needham-Schroeder property violation found by Tamarin

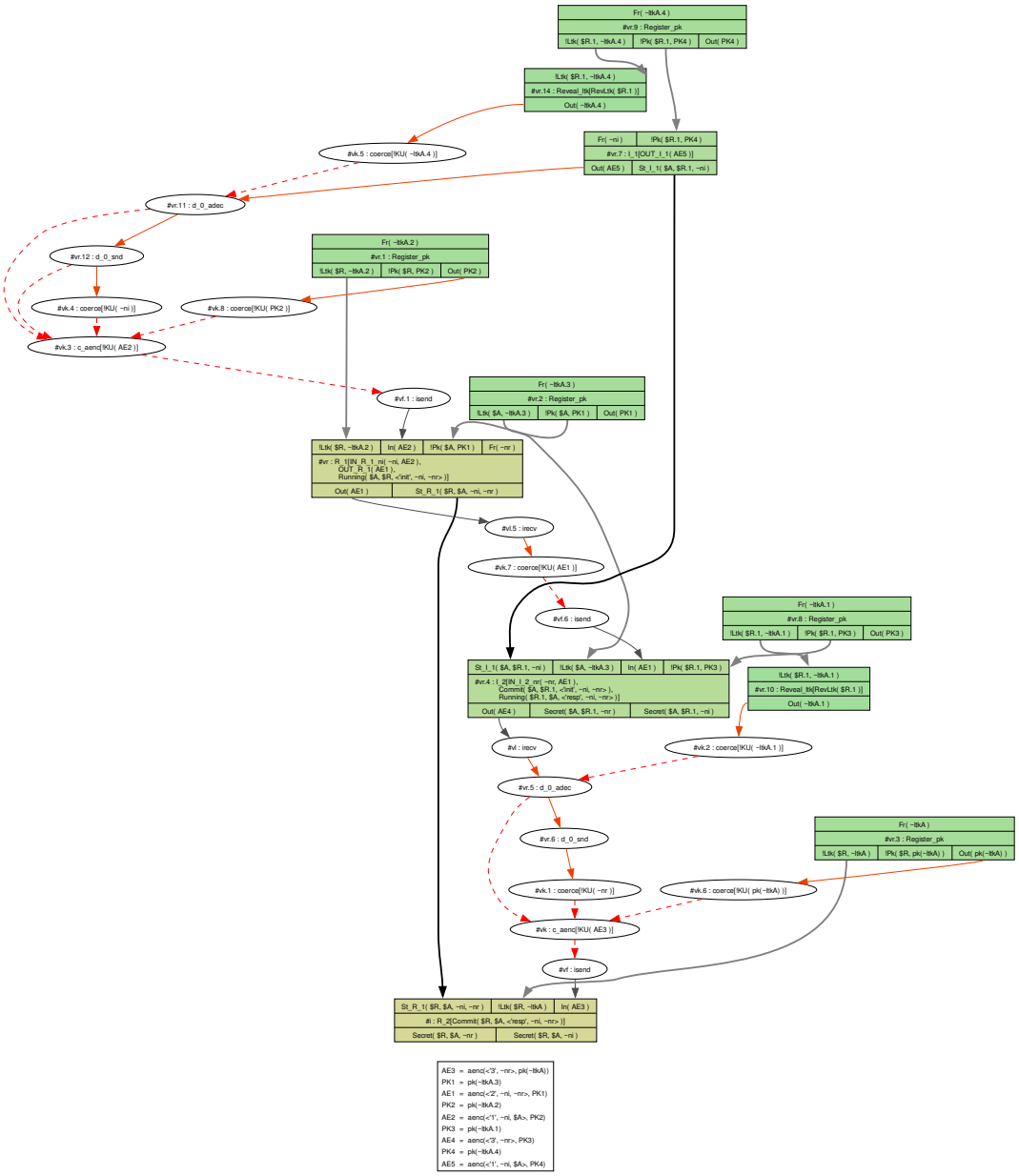

Figure 25: Second Needham-Schroeder property violation found by Tamarin

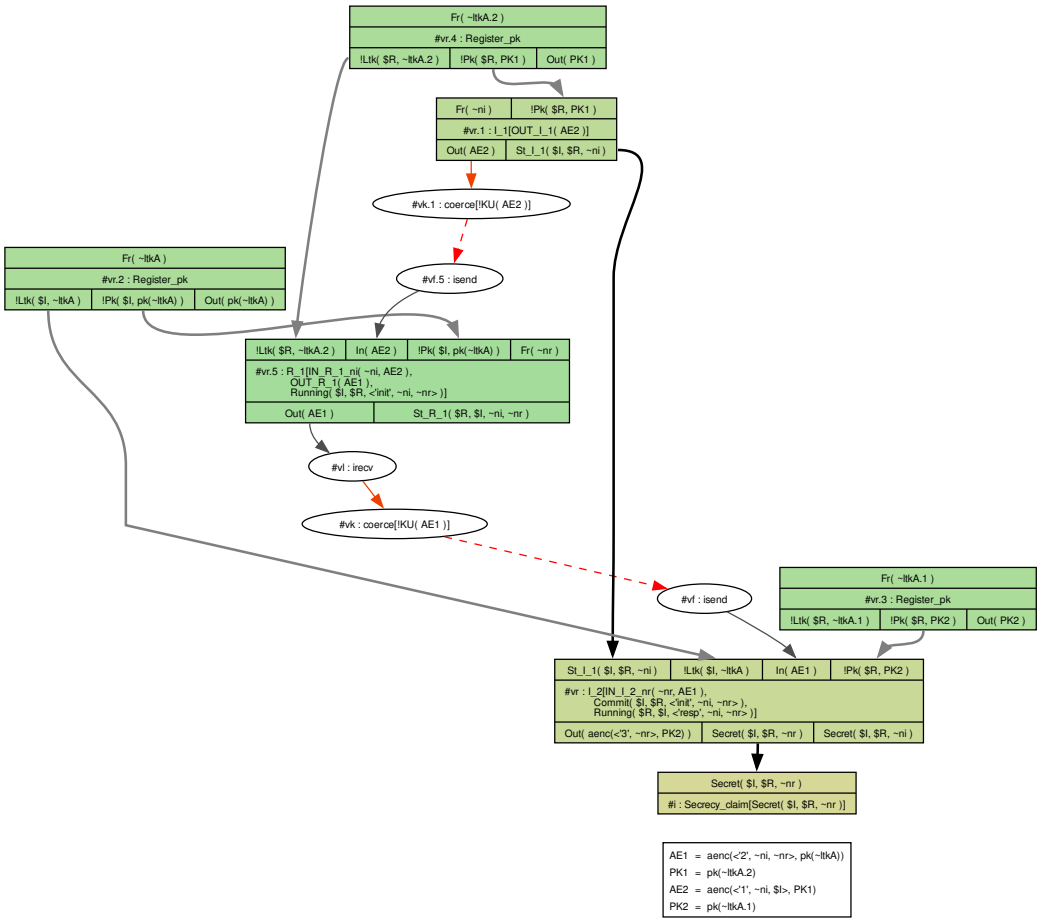

Figure 26: Third Needham-Schroeder property violation found by Tamarin

Listing 50: Needham-Schroeder with Lowe's fix in Tamarin

```
1:  theory NSPK3Fixed
2:  begin
3:
4:  builtins: asymmetric-encryption
5:
6:  /*
7:      Protocol:     The classic three message version of the
8:                    flawed Needham-Schroeder Public Key Protocol
9:      Modeler:      Simon Meier
10:     Date:         September 2012
11:
12:     Source:       Gavin Lowe. Breaking and fixing the
           Needham-Schroeder
```

```
13:                 public-key protocol using FDR. In Tiziana Margaria
                       and
14:                 Bernhard Steffen, editors, TACAS, volume 1055 of
                       Lecture Notes
15:                 in Computer Science, pages 147-166.  Springer,
                       1996.
16:
17:    Status:      working
18:
19:    Note that we are using explicit global constants for discerning
          the
20:    different encryption instead of the implicit sources.
21:  */
22:
23:
24:  // Public key infrastructure
25:  rule Register_pk:
26:    [ Fr(~ltkA) ]
27:    -->
28:    [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]
29:
30:  rule Reveal_ltk:
31:    [ !Ltk(A, ltkA) ] --[ RevLtk(A)    ]-> [ Out(ltkA) ]
32:
33:
34:  /* We formalize the following protocol
35:
36:    protocol NSPK3 {
37:      1. I -> R: {'1',ni,I}pk(R)
38:      2. I <- R: {'2',ni,nr,R}pk(I)
39:      3. I -> R: {'3',nr}pk(R)
40:    }
41:  */
42:
43:  rule I_1:
44:    let m1 = aenc{'1', ~ni, $I}pkR
45:    in
46:      [ Fr(~ni)
47:      , !Pk($R, pkR)
48:      ]
49:    --[ OUT_I_1(m1)
50:      ]->
51:      [ Out( m1 )
52:      , St_I_1($I, $R, ~ni)
53:      ]
54:
55:  rule R_1:
56:    let m1 = aenc{'1', ni, I}pk(ltkR)
57:        m2 = aenc{'2', ni, ~nr, $R}pkI
```

```
58:   in
59:     [ !Ltk($R, ltkR)
60:     , In( m1 )
61:     , !Pk(I, pkI)
62:     , Fr(~nr)
63:     ]
64:   --[ IN_R_1_ni( ni, m1 )
65:     , OUT_R_1( m2 )
66:     , Running(I, $R, <'init',ni,~nr>)
67:     ]->
68:     [ Out( m2 )
69:     , St_R_1($R, I, ni, ~nr)
70:     ]
71:
72: rule I_2:
73:   let m2 = aenc{'2', ni, nr, R}pk(ltkI)
74:       m3 = aenc{'3', nr}pkR
75:   in
76:     [ St_I_1(I, R, ni)
77:     , !Ltk(I, ltkI)
78:     , In( m2 )
79:     , !Pk(R, pkR)
80:     ]
81:   --[ IN_I_2_nr( nr, m2)
82:     , Commit (I, R, <'init',ni,nr>)  // need to log identities
          explicitely to
83:     , Running(R, I, <'resp',ni,nr>)  // specify that they must not
          be
84:                                      // compromised in the
                                            property.
85:     ]->
86:     [ Out( m3 )
87:     , Secret(I,R,nr)
88:     , Secret(I,R,ni)
89:     ]
90:
91: rule R_2:
92:     [ St_R_1(R, I, ni, nr)
93:     , !Ltk(R, ltkR)
94:     , In( aenc{'3', nr}pk(ltkR) )
95:     ]
96:   --[ Commit (R, I, <'resp',ni,nr>)
97:     ]->
98:     [ Secret(R,I,nr)
99:     , Secret(R,I,ni)
100:    ]
101:
102: /* TODO: Also model session-key reveals and adapt security
       properties. */
```

```
103: rule Secrecy_claim:
104:   [ Secret(A, B, m) ] --[ Secret(A, B, m) ]-> []
105:
106:
107:
108: /* Note that we are using an untyped protocol model. For proofs,
       we therefore
109: require a protocol specific type invariant for proof construction.
       In
110: principle, such an invariant is not required for attack search,
       but does help
111: a lot.
112:
113: See 'NSLPK3.spthy' for a detailed explanation of the construction
       of this
114: invariant.
115: */
116: lemma types [sources]:
117:   " (All ni m1 #i.
118:        IN_R_1_ni( ni, m1) @ i
119:        ==>
120:        ( (Ex #j. KU(ni) @ j & j < i)
121:        | (Ex #j. OUT_I_1( m1 ) @ j)
122:        )
123:     )
124:   & (All nr m2 #i.
125:        IN_I_2_nr( nr, m2) @ i
126:        ==>
127:        ( (Ex #j. KU(nr) @ j & j < i)
128:        | (Ex #j. OUT_R_1( m2 ) @ j)
129:        )
130:     )
131:   "
132:
133: // Nonce secrecy from the perspective of both the initiator and
       the responder.
134: lemma nonce_secrecy:
135:   " /* It cannot be that */
136:     not(
137:         Ex A B s #i.
138:           /* somebody claims to have setup a shared secret, */
139:           Secret(A, B, s) @ i
140:           /* but the adversary knows it */
141:         & (Ex #j. K(s) @ j)
142:           /* without having performed a long-term key reveal. */
143:         & not (Ex #r. RevLtk(A) @ r)
144:         & not (Ex #r. RevLtk(B) @ r)
145:        )"
146:
```

```
147: // Injective agreement from the perspective of both the initiator
       and the responder.
148: lemma injective_agree:
149:   " /* Whenever somebody commits to running a session, then*/
150:     All actor peer params #i.
151:         Commit(actor, peer, params) @ i
152:       ==>
153:         /* there is somebody running a session with the same
                parameters */
154:           (Ex #j. Running(actor, peer, params) @ j & j < i
155:             /* and there is no other commit on the same parameters
                   */
156:             & not(Ex actor2 peer2 #i2.
157:                     Commit(actor2, peer2, params) @ i2 & not(#i =
                           #i2)
158:                   )
159:           )
160:         /* or the adversary perform a long-term key reveal on
                actor or peer */
161:         | (Ex #r. RevLtk(actor) @ r)
162:         | (Ex #r. RevLtk(peer)  @ r)
163:   "
164:
165: // Consistency check: ensure that secrets can be shared between
       honest agents.
166: lemma session_key_setup_possible:
167:   exists-trace
168:   " /* It is possible that */
169:     Ex A B s #i.
170:       /* somebody claims to have setup a shared secret, */
171:       Secret(A, B, s) @ i
172:       /* without the adversary having performed a long-term key
              reveal. */
173:     & not (Ex #r. RevLtk(A) @ r)
174:     & not (Ex #r. RevLtk(B) @ r)
175:   "
176:
177: end
```

B.5 AVISPA

Listing 51: Needham-Schroeder in AVISPA

```
1: role alice (A, B: agent,
2:             Ka, Kb: public_key,
3:             SND, RCV: channel (dy))
4: played_by A def=
5:
```

```
6:    local State : nat,
7:          Na, Nb: text
8:
9:    init State := 0
10:
11:   transition
12:
13:     0.  State  = 0 /\ RCV(start) =|>
14:         State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)
15:                    /\ secret(Na',na,{A,B})
16:
17:
18:     2.  State  = 2 /\ RCV({Na.Nb'}_Ka) =|>
19:         State':= 4 /\ SND({Nb'}_Kb)
20:
21:                    % Alice checks that the received Na corresponds
                          to the nonce she sent earlier
22:                    /\ request(A,B,alice_bob_na,Na)
23:
24:                    % Alice sends Nb to prove her identity
25:                    /\ witness(A,B,bob_alice_nb,Nb')
26:
27: end role
28:
29: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30:
31: role bob(A, B: agent,
32:          Ka, Kb: public_key,
33:          SND, RCV: channel (dy))
34: played_by B def=
35:
36:   local State : nat,
37:         Na, Nb: text
38:
39:   init State := 1
40:
41:   transition
42:
43:     1.  State  = 1 /\ RCV({Na'.A}_Kb) =|>
44:         State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
45:                    /\ secret(Nb',nb,{A,B})
46:                    % Bob sends Na to prove his identity
47:                    /\ witness(B,A,alice_bob_na,Na')
48:
49:     3.  State  = 3 /\ RCV({Nb}_Kb) =|>
50:                    % Bob checks that the received Nb corresponds
                          to the nonce he sent earlier
51:         State':= 5 /\ request(B,A,bob_alice_nb,Nb)
52:
```

```
53: end role
54:
55: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
56:
57: role session(A, B: agent, Ka, Kb: public_key) def=
58:
59:   local SA, RA, SB, RB: channel (dy)
60:
61:   composition
62:
63:     alice(A,B,Ka,Kb,SA,RA)
64:      /\ bob  (A,B,Ka,Kb,SB,RB)
65:
66: end role
67:
68: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
69:
70: role environment() def=
71:
72:     const a, b          : agent,
73:           ka, kb, ki    : public_key,
74:           na, nb,
75:           alice_bob_na,
76:           bob_alice_nb : protocol_id
77:
78:     intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}
79:
80:     composition
81:
82:         session(a,b,ka,kb)
83:  %     /\  session(a,b,ka,kb)
84:  %    /\ session(b,a,kb,ka)
85:  %     /\ session(i,b,ki,kb)
86:       /\ session(a,i,ka,ki)
87:
88:
89: end role
90:
91: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
92:
93: goal
94:
95:   secrecy_of na, nb
96:   authentication_on alice_bob_na
97:   authentication_on bob_alice_nb
98:
99: end goal
100:
101: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
102:
103: environment()
```

Listing 52: Needham-Schroeder with Lowe's fix in AVISPA

```
1:  role alice (A, B: agent,
2:              Ka, Kb: public_key,
3:              SND, RCV: channel (dy))
4:  played_by A def=
5:
6:    local State : nat,
7:          Na, Nb: text
8:
9:    init State := 0
10:
11:   transition
12:
13:     0.  State  = 0 /\ RCV(start) =|>
14:         State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)
15:                    /\ secret(Na',na,{A,B})
16:
17:
18:     2.  State  = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
19:         State':= 4 /\ SND({Nb'}_Kb)
20:
21:                    % Alice checks that the received Na corresponds
                          to the nonce she sent earlier
22:                    /\ request(A,B,alice_bob_na,Na)
23:
24:                    % Alice sends Nb to prove her identity
25:                    /\ witness(A,B,bob_alice_nb,Nb')
26:
27: end role
28:
29: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30:
31: role bob(A, B: agent,
32:           Ka, Kb: public_key,
33:           SND, RCV: channel (dy))
34: played_by B def=
35:
36:   local State : nat,
37:         Na, Nb: text
38:
39:   init State := 1
40:
41:   transition
42:
43:     1.  State  = 1 /\ RCV({Na'.A}_Kb) =|>
44:         State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'.B}_Ka)
```

```
                     /\ secret(Nb',nb,{A,B})
                     % Bob sends Na to prove his identity
                     /\ witness(B,A,alice_bob_na,Na')

    3.  State  = 3 /\ RCV({Nb}_Kb) =|>
                     % Bob checks that the received Nb corresponds
                        to the nonce he sent earlier
        State':= 5 /\ request(B,A,bob_alice_nb,Nb)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role session(A, B: agent, Ka, Kb: public_key) def=

  local SA, RA, SB, RB: channel (dy)

  composition

    alice(A,B,Ka,Kb,SA,RA)
     /\ bob  (A,B,Ka,Kb,SB,RB)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role environment() def=

    const a, b          : agent,
          ka, kb, ki    : public_key,
          na, nb,
          alice_bob_na,
          bob_alice_nb : protocol_id

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition

        session(a,b,ka,kb)
 %     /\  session(a,b,ka,kb)
 %    /\ session(b,a,kb,ka)
 %     /\ session(i,b,ki,kb)
      /\ session(a,i,ka,ki)


end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
93:  goal
94:
95:    secrecy_of na, nb
96:    authentication_on alice_bob_na
97:    authentication_on bob_alice_nb
98:
99:  end goal
100:
101: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
102:
103: environment()
```

References

[Abr+06] Jean-Raymond Abrial et al. "An Open Extensible Tool Environment for Event-B". In: *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 2006, pp. 588–605. ISBN: 9783540474623. DOI: 10.1007/11901433_32. URL: http://dx.doi.org/10.1007/11901433_32 (cit. on p. 4).

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 2010 (cit. on p. 4).

[Abr74] Jean-Raymond Abrial. "Data Semantics". In: *Proceedings of the IFIP Working Conference on Data Base Management*. North-Holland, 1974, pp. 1–59 (cit. on p. 5).

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996 (cit. on p. 4).

[AK24] Julius Armbrüster and Philipp Körner. "Meta-programming Event-B: Advancing Tool Support and Language Extensions". In: *Rigorous State-Based Methods*. Springer Nature Switzerland, 2024, pp. 233–240. ISBN: 9783031637902. DOI: 10.1007/978-3-031-63790-2_17. URL: http://dx.doi.org/10.1007/978-3-031-63790-2_17 (cit. on pp. 8, 51).

[BCM18] David Basin, Cas Cremers, and Catherine Meadows. "Model Checking Security Protocols". In: *Handbook of Model Checking*. Springer International Publishing, 2018, pp. 727–762. ISBN: 9783319105758. DOI: 10.1007/978-3-319-10575-8_22. URL: http://dx.doi.org/10.1007/978-3-319-10575-8_22 (cit. on p. 10).

[Ben+21] Jens Bendisposto et al. "ProB2-UI: A Java-Based User Interface for ProB". In: *Formal Methods for Industrial Critical Systems*. Springer International Publishing, 2021, pp. 193–201. ISBN: 9783030852481. DOI: 10.1007/978-3-030-85248-1_12. URL: http://dx.doi.org/10.1007/978-3-030-85248-1_12 (cit. on p. 5).

[Ber08] D. J. Bernstein. *ChaCha, a variant of Salsa20*. Tech. rep. Jan. 2008. URL: http://cr.yp.to/chacha.html (cit. on p. 12).

[Bla01] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society, June 2001, pp. 82–96. DOI: 10.1109/csfw.2001.930138 (cit. on p. 2).

[BM10] Nazim Benaissa and Dominique Méry. "Cryptographic Protocols Analysis in Event B". In: *Perspectives of Systems Informatics*. Springer Berlin Heidelberg, 2010, pp. 282–293. ISBN: 9783642114861. DOI: 10.1007/978-3-642-11486-1_24. URL: http://dx.doi.org/10.1007/978-3-642-11486-1_24 (cit. on p. 4).

[BM13] Michael Butler and Issam Maamria. "Practical Theory Extension in Event-B". In: *Theories of Programming and Formal Methods*. Springer Berlin Heidelberg, 2013, pp. 67–81. ISBN: 9783642396984. DOI: 10.1007/978-3-642-39698-4_5. URL: http://dx.doi.org/10.1007/978-3-642-39698-4_5 (cit. on pp. 5, 52).

[Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017. DOI: 10.17487/rfc8259. URL: https://www.rfc-editor.org/rfc/rfc8259 (cit. on p. 8).

[But+20] Michael Butler et al. "The First Twenty-Five Years of Industrial Use of the B-Method". In: *Formal Methods for Industrial Critical Systems*. Springer International Publishing, 2020, pp. 189–209. ISBN: 9783030582982. DOI: 10.1007/978-3-030-58298-2_8. URL: http://dx.doi.org/10.1007/978-3-030-58298-2_8 (cit. on p. 4).

[CC04] Hubert Comon-Lundh and Véronique Cortier. "Security properties: two agents are sufficient". In: *Science of Computer Programming* 50.1–3 (Mar. 2004), pp. 51–71. ISSN: 0167-6423. DOI: 10.1016/j.scico.2003.12.002. URL: http://dx.doi.org/10.1016/j.scico.2003.12.002 (cit. on pp. 66, 68).

[Cla+99] E. Clarke et al. *Model Checking*. MIT Press, 1999 (cit. on pp. 64, 66).

[CM11] Mats Carlsson and Per Mildner. "SICStus Prolog – The first 25 years". In: *Theory and Practice of Logic Programming* 12.1–2 (Sept. 2011), pp. 35–66. ISSN: 1475-3081. DOI: 10.1017/s1471068411000482. URL: http://dx.doi.org/10.1017/s1471068411000482 (cit. on p. 5).

[Cre08] C. J. F. Cremers. "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.* Vol. 5123/2008. Lecture Notes in Computer Science. Springer, 2008, pp. 414–418. DOI: 10.1007/978-3-540-70545-1_38 (cit. on p. 2).

[DEK82] D. Dolev, S. Even, and R.M. Karp. "On the security of ping-pong protocols". In: *Information and Control* 55.1–3 (Oct. 1982), pp. 57–68. ISSN: 0019-9958. DOI: 10.1016/s0019-9958(82)90401-6. URL: http://dx.doi.org/10.1016/s0019-9958(82)90401-6 (cit. on p. 10).

[DY83] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (Mar. 1983), pp. 198–208. ISSN: 0018-9448. DOI: 10.1109/tit.1983.1056650. URL: http://dx.doi.org/10.1109/tit.1983.1056650 (cit. on p. 10).

[Ecm25] Ecma. *ECMAScript® 2025 language specification*. ECMA 262. Ecma, June 2025. URL: https://262.ecma-international.org/ (cit. on p. 8).

[For89] Internet Engineering Task Force. *Requirements for Internet Hosts – Communication Layers*. RFC 1122. RFC Editor, Oct. 1989. DOI: 10.17487/rfc1122. URL: https://www.rfc-editor.org/info/rfc1122 (cit. on p. 1).

[Glo+06] Y. Glouche et al. "A Security Protocol Animator Tool for AVISPA". In: *ARTIST-2 workshop on security of embedded systems, Pisa (Italy)*. 2006. URL: http://people.irisa.fr/Thomas.Genet/span/ (cit. on p. 3).

[Gos+25] James Gosling et al. *The Java Language Specification: Java SE 24 Edition*. Oracle America, Inc. Mar. 2025. URL: https://docs.oracle.com/javase/specs/jls/se24/jls24.pdf (visited on 2025-07-08) (cit. on p. 5).

[Hic12] Rich Hickey. *Extensible Data Notation (EDN)*. 2012. URL: https://github.com/edn-format/edn (visited on 2025-07-08) (cit. on p. 8).

[Hic20] Rich Hickey. "A history of Clojure". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (June 2020), pp. 1–46. ISSN: 2475-1421. DOI: 10.1145/3386321. URL: http://dx.doi.org/10.1145/3386321 (cit. on p. 6).

[HL12] Dominik Hansen and Michael Leuschel. "Translating TLA$^+$ to B for Validation with ProB". In: *Integrated Formal Methods*. Ed. by John Derrick et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 24–38. ISBN: 978-3-642-30729-4 (cit. on p. 6).

[HL14] Dominik Hansen and Michael Leuschel. "Translating B to TLA$^+$ for Validation with TLC". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 40–55. ISBN: 978-3-662-43652-3 (cit. on pp. 6, 53).

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985 (cit. on p. 82).

[HSL16] Dominik Hansen, David Schneider, and Michael Leuschel. "Using B and ProB for Data Validation Projects". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer International Publishing, 2016, pp. 167–182. ISBN: 9783319336008. DOI: 10.1007/978-3-319-33600-8_10. URL: http://dx.doi.org/10.1007/978-3-319-33600-8_10 (cit. on p. 4).

[ISO02] ISO/IEC. *Information technology – Z formal specification notation – Syntax, type system and semantics*. Standard ISO/IEC 13568:2002. Geneva, CH: International Organization for Standardization, July 2002. URL: https://www.iso.org/standard/21573.html (cit. on p. 5).

[Kan+15] Gijs Kant et al. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 9783662466810. DOI: 10.1007/978-3-662-46681-0_61. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_61 (cit. on pp. 60, 70).

[Kau+14] C. Kaufman et al. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. RFC Editor, Oct. 2014. DOI: 10.17487/rfc7296. URL: https://www.rfc-editor.org/info/rfc7296 (cit. on p. 1).

[KL23] Philipp Körner and Michael Leuschel. "Towards Practical Partial Order Reduction for High-Level Formalisms". In: *Verified Software. Theories, Tools and Experiments*. Springer International Publishing, 2023, pp. 72–91. ISBN: 9783031258039. DOI: 10.1007/978-3-031-25803-9_5. URL: http://dx.doi.org/10.1007/978-3-031-25803-9_5 (cit. on p. 64).

[KLM18] Philipp Körner, Michael Leuschel, and Jeroen Meijer. "State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin". In: *Integrated Formal Methods*. Springer International Publishing, 2018, pp. 275–295. ISBN: 9783319989389. DOI: 10.1007/978-3-319-98938-9_16. URL: http://dx.doi.org/10.1007/978-3-319-98938-9_16 (cit. on p. 70).

[KM22] Philipp Körner and Florian Mager. "An embedding of B in Clojure". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. ACM, Oct. 2022, pp. 598–606. DOI: 10.1145/3550356.3561561. URL: http://dx.doi.org/10.1145/3550356.3561561 (cit. on p. 8).

[KMR25] Philipp Körner, Florian Mager, and Jan Roßbach. "A case for data-oriented specifications: simpler implementation of B tools and DSLs". In: *Innovations in Systems and Software Engineering* (Mar. 2025). ISSN: 1614-5054. DOI: 10.1007/s11334-025-00596-3. URL: http://dx.doi.org/10.1007/s11334-025-00596-3 (cit. on p. 8).

[Kob87] Neal Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (Jan. 1987), p. 203. ISSN: 0025-5718. DOI: 10.2307/2007884. URL: http://dx.doi.org/10.2307/2007884 (cit. on p. 12).

[Kör+20] Philipp Körner et al. "Integrating formal specifications into applications: the ProB Java API". In: *Formal Methods in System Design* 58.1-2 (Oct. 2020), pp. 160–187. ISSN: 1572-8102. DOI: 10.1007/s10703-020-00351-3. URL: http://dx.doi.org/10.1007/s10703-020-00351-3 (cit. on pp. 5, 7).

[Lam99] Leslie Lamport. "Specifying Concurrent Systems with TLA+". In: *Calculational System Design* (Apr. 1999), pp. 183–247. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/ (cit. on pp. 5, 6, 60).

[LB03] Michael Leuschel and Michael Butler. "ProB: A Model Checker for B". In: *FME 2003: Formal Methods*. Vol. 2805. LNCS. Berlin, Heidelberg: Springer, Sept. 2003, pp. 855–874 (cit. on p. 5).

[LB08] Michael Leuschel and Michael Butler. "ProB: An Automated Analysis Toolset for the B Method". In: *International Journal on Software Tools for Technology Transfer* 10.2 (Mar. 2008), pp. 185–203 (cit. on p. 5).

[Leu+06] Michael Leuschel et al. "Symmetry Reduction for B by Permutation Flooding". In: *B 2007: Formal Specification and Development in B*. Springer Berlin Heidelberg, 2006, pp. 79–93. ISBN: 9783540687610. DOI: 10.1007/11955757_9. URL: http://dx.doi.org/10.1007/11955757_9 (cit. on pp. 66, 67).

[Leu01] Michael Leuschel. "Design and Implementation of the High-Level Specification Language CSP(LP) in Prolog". In: *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2001, pp. 14–28. ISBN: 9783540452416. DOI: 10.1007/3-540-45241-9_2. URL: http://dx.doi.org/10.1007/3-540-45241-9_2 (cit. on p. 82).

[Leu21] Michael Leuschel. "Spot the Difference: A Detailed Comparison Between B and Event-B". In: *Logic, Computation and Rigorous Methods*. Springer International Publishing, 2021, pp. 147–172. ISBN: 9783030760205. DOI: 10.1007/978-3-030-76020-5_9. URL: http://dx.doi.org/10.1007/978-3-030-76020-5_9 (cit. on p. 4).

[Lin+25] Tim Lindholm et al. *The Java Virtual Machine Specification: Java SE 24 Edition*. Oracle America, Inc. Mar. 2025. URL: https://docs.oracle.com/javase/specs/jvms/se24/jvms24.pdf (visited on 2025-07-08) (cit. on p. 7).

[LM00] Michael Leuschel and Thierry Massart. "Infinite State Model Checking by Abstract Interpretation and Program Specialisation". In: *Logic-Based Program Synthesis and Transformation*. Springer Berlin Heidelberg, 2000, pp. 62–81. ISBN: 9783540451488. DOI: 10.1007/10720327_5. URL: http://dx.doi.org/10.1007/10720327_5 (cit. on p. 5).

[LM10] Michael Leuschel and Thierry Massart. "Efficient approximate verification of B and Z models via symmetry markers". In: *Annals of Mathematics and Artificial Intelligence* 59.1 (May 2010), pp. 81–106. ISSN: 1573-7470. DOI: 10.1007/s10472-010-9208-8. URL: http://dx.doi.org/10.1007/s10472-010-9208-8 (cit. on pp. 66, 67).

[Low95] Gavin Lowe. "An attack on the Needham-Schroeder public-key authentication protocol". In: *Information Processing Letters* 56.3 (Nov. 1995), pp. 131–133. ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00144-2. URL: http://dx.doi.org/10.1016/0020-0190(95)00144-2 (cit. on p. 71).

[Low96] Gavin Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 1996, pp. 147–166. ISBN: 9783540498742. DOI: 10.1007/3-540-61042-1_43. URL: http://dx.doi.org/10.1007/3-540-61042-1_43 (cit. on p. 71).

[Low97] G. Lowe. "A hierarchy of authentication specifications". In: *Proceedings 10th Computer Security Foundations Workshop*. CSFW-97. IEEE Comput. Soc. Press, 1997, pp. 31–43. DOI: 10.1109/csfw.1997.596782. URL: http://dx.doi.org/10.1109/csfw.1997.596782 (cit. on p. 18).

[McC60] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199 (cit. on p. 6).

[Mei+13] Simon Meier et al. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc.* Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48 (cit. on p. 2).

[Mil86] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology - CRYPTO '85 Proceedings*. Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 9783540164630. DOI: 10.1007/3-540-39799-x_31. URL: http://dx.doi.org/10.1007/3-540-39799-x_31 (cit. on p. 12).

[Neu+05] C. Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120. RFC Editor, July 2005. DOI: 10.17487/rfc4120. URL: https://www.rfc-editor.org/rfc/rfc4120 (cit. on p. 1).

[NIS23] NIST. *Advanced Encryption Standard (AES)*. FIPS 197. National Institute of Standards and Technology (U.S.), May 2023. DOI: 10.6028/nist.fips.197-upd1. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf (cit. on p. 12).

[NS78] Roger M. Needham and Michael D. Schroeder. "Using encryption for authentication in large networks of computers". In: *Communications of the ACM* 21.12 (Dec. 1978), pp. 993–999. ISSN: 1557-7317. DOI: 10.1145/359657.359659. URL: http://dx.doi.org/10.1145/359657.359659 (cit. on p. 71).

[PL07] Daniel Plagge and Michael Leuschel. "Validating Z Specifications Using the ProB Animator and Model Checker". In: *Integrated Formal Methods*. Springer Berlin Heidelberg, 2007, pp. 480–500. ISBN: 9783540732105. DOI: 10.1007/978-3-540-73210-5_25. URL: http://dx.doi.org/10.1007/978-3-540-73210-5_25 (cit. on p. 5).

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018. DOI: 10.17487/rfc8446. URL: https://www.rfc-editor.org/rfc/rfc8446 (cit. on p. 1).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 1557-7317. DOI: 10.1145/359340.359342. URL: http://dx.doi.org/10.1145/359340.359342 (cit. on p. 12).

[Sch15] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. Wiley, Oct. 2015. ISBN: 9781119183471. DOI: 10.1002/9781119183471. URL: http://dx.doi.org/10.1002/9781119183471 (cit. on p. 33).

[SL08] Corinna Spermann and Michael Leuschel. "ProB gets Nauty: Effective Symmetry Reduction for B and Z Models". In: *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, June 2008, pp. 15–22. DOI: 10.1109/tase.2008.33. URL: http://dx.doi.org/10.1109/tase.2008.33 (cit. on p. 67).

[VBL22] Fabian Vu, Dominik Brandt, and Michael Leuschel. "Model Checking B Models via High-Level Code Generation". In: *Formal Methods and Software Engineering*. Springer International Publishing, 2022, pp. 334–351. ISBN: 9783031172441. DOI: 10.1007/978-3-031-17244-1_20. URL: http://dx.doi.org/10.1007/978-3-031-17244-1_20 (cit. on p. 82).

[Vig06] Luca Viganò. "Automated Security Protocol Analysis With the AVISPA Tool". In: *Electronic Notes in Theoretical Computer Science* 155 (May 2006), pp. 61–86. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.11.052. URL: http://dx.doi.org/10.1016/j.entcs.2005.11.052 (cit. on p. 3).

[Vu+19] Fabian Vu et al. "A Multi-target Code Generator for High-Level B". In: *Integrated Formal Methods*. Springer International Publishing, 2019, pp. 456–473. ISBN: 9783030349684. DOI: 10.1007/978-3-030-34968-4_25. URL: http://dx.doi.org/10.1007/978-3-030-34968-4_25 (cit. on p. 82).

[WL20] Michelle Werth and Michael Leuschel. "VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics". In: *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*. Ed. by Alexander Raschke, Dominique Méry, and Frank Houdek. Vol. 12071. Lecture Notes in Computer Science. Springer, 2020, pp. 260–265. DOI: 10.1007/978-3-030-48077-6_21 (cit. on p. 57).

[Ylo06] T. Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. RFC Editor, Jan. 2006. DOI: 10.17487/rfc4253. URL: https://www.rfc-editor.org/rfc/rfc4253 (cit. on p. 1).

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. CHARME '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 54–66. ISBN: 3540665595. DOI: 10.1007/3-540-48153-2_6 (cit. on p. 6).