HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Laboratory for Theoretical Computer Science

Antti Huima

Analysis of cryptographic protocols via symbolic state space enumeration

Master's Thesis
May 17, 1999

Supervisor Professor Leo Ojala
Instructor Lic. Tech. Tatu Ylönen

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Antti Huima		
Name of the thesis:	Analysis of cryptographic protocols via symbolic state space enumeration		
Date:	May 17, 1999	**Pages:** 13+142	
Department:	Faculty of Information Technology	**Professorship:** Tik-79	
Supervisor:	Professor Leo Ojala		
Instructor:	Lic. Tech. Tatu Ylönen		

Cryptographic protocols are an important building block of information security solutions.

Many proposed security protocols have been found to have defects in their logical structures. Such defects can cause the protocol to be vulnerable to various kinds of attacks. In the worst case, a faulty protocol can cause large monetary and other losses. Therefore many methods for analyzing security protocols formally have appeared in the scientific literature.

In this thesis, a new formal analysis method for security protocols is presented. The analysis method is based on symbolic state space enumeration. Symbolic state space enumeration is a generic concept in the verification of concurrent systems. This work applies the concept to the domain of security protocols.

Symbolic state space enumeration makes it possible to search an infinite state space in finite time.

Cryptography is modeled using algebras and convergent rewrite systems. Typically, models involving rewrite systems are implemented by using variants of the narrowing algorithm. However, the presented method does not need to use narrowing because of a novel, alternative approach bound tightly to the whole search process.

In the case of formal verification of generic concurrent systems, much of the data flow can be abstracted away. However, for security protocols, data is of the essence. Most of the complexity of the symbolic method in this thesis stems from the need to be able to handle infinite sets of algebraic terms and their interactions under a rewrite system.

In addition to the basic symbolic method, a symmetry method for the symbolic states as well as a method for detecting symbolic state subsumption relations is developed.

An end-user tool based on the theory, implemented in Scheme, is described.

As a whole the work is divided into two parts. In the first part the theoretical aspects of the method are described. The second part focuses on the implementation; also, an example is presented there.

Keywords:	cryptographic protocols, formal methods, state space enumeration, symbolic methods
Language:	English

TEKNILLINEN KORKEAKOULU — DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Antti Huima		
Työn nimi:	Kryptografisten protokollien analysointi symbolisella tilahaulla		
Päivämäärä:	17. toukokuuta 1999	**Sivuja:**	13+142
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-79
Työn valvoja:	Professori Leo Ojala		
Työn ohjaaja:	Tekn. lis. Tatu Ylönen		

Kryptografiset protokollat ovat keskeisessä asemassa tietoturvaratkaisuja kehitettäessä.

Monet suunnitelluista turvallisuusprotokollista on havaittu myöhemmin rakenteeltaan virheellisiksi. Virheet protokollan loogisessa rakenteessa voivat mahdollistaa erilaisia hyökkäyksiä protokollaa vastaan. Pahimmassa tapauksessa tuloksena ovat suuret taloudelliset ja muunlaiset menetykset. Tästä syystä tieteellisessä kirjallisuudessa onkin esitetty useita menetelmiä kryptografisten protokollien formaalia verifiointia varten.

Tässä työssä esitetään uusi formaali menetelmä tietoturvaprotokollien analysoimiseksi. Analyysimenetelmä perustuu symbolisen tilahakuun. Symbolinen tilahaku itsessään on yleiskäsite joka viittaa luokkaan rinnakkaisten järjestelmien käyttäytymistä tutkittaessa käytettäviä menetelmiä.

Symbolinen tilahaku mahdollistaa äärettömän tila-avaruuden läpikäymisen äärellisessä ajassa.

Kryptografisia operaatioita mallinnetaan työssä algebrojen ja uudelleenkirjoitusjärjestelmien avulla. Yleensä mallit jotka käyttävät uudelleenkirjoitusjärjestelmiä toteutetaan narrowing-algoritmin muunnelmilla. Tässä työssä narrowing-algoritmia ei tarvita uuden, vaihtoehtoisen lähestymistavan vuoksi joka on sidottu kiinteästi koko hakuprosessiin.

Rinnakkaisten järjestelmien formaalissa analyysissa voidaan tyypillisesti suurin osa datasta abstrahoida pois. Tietoturvaprotokollien kohdalla tilanne on toinen, koska nimenomaan siirretyt viestit ovat kiinnostavia. Suurin osa esitellyn symbolisen menetelmän monimutkaisuudesta johtuukin tarpeesta käsitellä äärettömiä termijoukkoja ja niiden välisiä suhteita uudelleenkirjoitusjärjestelmien alaisuudessa.

Perusmenetelmän lisäksi työssä esitellään symmetriamenetelmä symbolisille tiloille ja menetelmä joidenkin symbolisten tilojen alisteisuussuhteiden havaitsemiseksi.

Analysointimenetelmän lisäksi myös menetelmästä tehty loppukäyttäjätoteutus esitellään. Ohjelmisto on toteutettu Scheme-ohjelmointikielellä.

Työ jakautuu kahteen osaan. Ensimmäinen osa esittää teoreettisen menetelmän. Toinen osa keskittyy toteutukseen ja sisältää esimerkin.

Avainsanat:	kryptografiset protokollat, formaalit menetelmät, tilahaku, symboliset menetelmät
Kieli:	englanti

Acknowledgments

First and foremost, I wish to thank SSH Communications Security, Ltd. for the monetary support I received, and especially the company's founder Tatu Ylönen who acted as my instructor during the preparation of this thesis.

I want to express my gratitude towards my professor Leo Ojala. He has had an open and very supportive attitude towards my studies in general as well as this particular work. The inspirational value of his esteem for formal rigor cannot be neglected.

I want to thank Tuomas Aura and Pekka Nikander for the support I have received from them. I acknowledge their influence in directing me towards the field of computer security.

Furthermore, I want to thank all the staff of the Laboratory for Theoretical Computer Science for their friendship and the exciting atmosphere of the laboratory that encourages formal rigor and exact mathematical treatment. Special thanks are due to Tommi Junttila.

This work was typeset using Donald Knuth's and Leslie Lamport's LaTeX system. I benefited from using Danny Dubé's SILex system for the tool implementation.

Otaniemi, May 1999

Antti Huima

Contents

List of Figures

List of Notation

$\underset{\mathfrak{R}_t}{\rightarrow}$, the top-level rewrite relation for $\mathfrak{R}$, 28
ρ, the representation function, 37, 57
$\mathfrak{R}$, the parameterized rewrite system, 22
$\mathfrak{R}$, the parametrised rewrite system, 23
$\mathfrak{R}(\mathscr{D})$, parametrised rewrite system with explicit parameter, 24

$\mathscr{S}$, the symbolic successor function, 81
s, a generic symbolic successor state function, 37
$\mathbf{safety}(\cdot,\cdot)$, predicate capturing a notion of safety, 35
$\cdot \mathbin{\S} \cdot$, the concatenation of two sequences or traces, 87, 113
$[\![\cdot]\!]$, semantical braces, 18
$[\![\cdot]\!]$, semantical braces in denotational semantics, 113
σ, substitution, 17
$\mathcal{S}$, signature, 17, 18
$\rightsquigarrow \cdot$, the maximal step transition relation, 85
$\underset{\cdot}{\rightsquigarrow}$, the labeled maximal step transition relation, 85
$\overset{s}{\rightsquigarrow} \cdot$, the maximal symbolic step transition relation, 87
$\underset{\cdot}{\overset{s}{\rightsquigarrow}}$, the labeled maximal symbolic step transition relation, 87
$\mathrm{sub}(\cdot)$, the set of all the subprograms of a program, 32
$\mathrm{sub}_0(\cdot)$, the set of the immediate subprograms of a program, 32
$\mathbf{subst}(\mathcal{X}, A)$, the set of substitutions from $\mathcal{X}$ to A, 17
$\lhd$, term and substitution more-general-than relation, 18
$\unlhd$, term and substitution subsumption relation, 18
$\preccurlyeq$, the symbolic state subsumption relation, 67
$\underset{\mathbf{sym}}{\approx}$, the symmetry relation, 88

T, the set of all the predicates defined over the terminal global states, 35
τ, a terminal predicate, 35
$\hat{\tau}$, a terminal predicate for symbolic states corresponding to τ, 37, 71
$\mathbf{TC}(\cdot)$, the algebraic closure under $\mathbf{F}(\mathcal{S})$, 48
$t, _p$, the subterm of t in the position p, an occurrence19
$\mathbf{T}_{\mathcal{S}}(Q)$, set of terms, 17
$t[s]_p$, the subterm in the position p replaced by s, 19
$\mathbf{T}_{\mathcal{S}}(\emptyset)$, ground terms, 18
R^+, the transitive closure of R, 15
$\rightarrow$, transition relation, 34
$\underset{\cdot}{\rightarrow}$, labeled transition relation, 34
R^*, the transitive and reflexive closure of R, 15
TRUE, element of $\mathscr{F}_0$, 24

$\mathrm{unify}(\cdot,\cdot)$, unification function, 18
$\tilde{\Upsilon}$, the function mapping terms containing constant-qualified state variables to ground terms, 38

Υ, a valuation, i.e. a function mapping state variables to constant terms, 34
$\hat{\tilde{\Upsilon}}$, the symbolic counterpart of $\tilde{\Upsilon}$, 74
$\hat{\Upsilon}$, a symbolic valuation, i.e. a function mapping state variables to terms containing variables, 56

$\mathcal{V}_{\mathcal{N}}$, set of constant-qualified state variables, 38
$\mathcal{V}_{\mathcal{N} \cup \mathscr{X}_P}$, set of qualified state variables, 38
$\mathcal{V}$, the set of state variables, 31

$\mathscr{X}_P$, set of variables ranging over participants, 38
$\mathcal{X}$, set of variables, 17

$\mathbb{Z}_+$, positive integers, i.e. integers > 0, 14

Chapter 1

Introduction

The general problem of analyzing cryptographic protocols is introduced and some current methodologies are surveyed. The scope of this thesis is defined.

This thesis is the culmination of our work done at SSH Communications Security from June '98 until the end of February '99.

The goal of the thesis project was to develop a software tool for automated verification of cryptographic protocols. The project was successful. In this thesis, we present the theory behind the implemented tool — theory was developed during this project — as well as some highlights of the software itself.

This thesis is divided into two parts. The first part describes the theoretical development. The second part discusses the implementation. The first part is independent of the second.

The structure of this thesis is the following. The rest of this chapter serves as a short introduction to the field of formal analysis of cryptographic protocols. The next chapter gives an overview of the project and the tool.

After that, the next few chapters constitute the theory part. The theory part starts with a chapter that gives prerequisites necessary for fully understanding the theory. Most of the prerequisites are quite standard formal machinery. We nevertheless suggest the reader to browse through the chapter, even if the reader is familiar with the constructs. This is because we fix the notation we use throughout the thesis there.

The remaining chapters of the theory part describe our formal model for cryptographic protocols. Also, the symbolic state space enumeration method for our problem is presented.

Proofs of results in the theoretical section are included in the body text. We see that they are essential for understanding the development of the theory. All proofs are not complete and some proofs are omitted.

The implementation part is divided into four chapters. The first chapter gives an overview of the implementation. The second chapter describes the higher-level language that is used to describe cryptographic protocols. In the same chapter we also show how it is translated into the formal model, which is the subject of the first part. Then a chapter about some details follows. The fourth chapter gives an example, using the Needham-Schröder Public-Key protocol as a case.

Conclusions are drawn in the last chapter.

1.1 Cryptographic Protocols and Their Analysis

Cryptographic protocols (security protocols) are communication protocols that employ *cryptographic algorithms* and whose primary purpose is to achieve *security-related* goals. These goals typically include the authentication of the communicating parties and the establishment of a secret session key.

Security protocols are necessary for many communications applications that require data protection and authentication. These applications include, for example, electronic commerce, tele-voting and intranet solutions. Therefore, security protocols have a major role in the information society. Slightly exaggerating, we can say that they form the foundations upon which the electronic society will be built.

1.1.1 What Does It Mean to Analyse a Security Protocol?

Cryptosystems, such as DES [68] or IDEA [42, 43], are commonly analyzed (by hand) in order to ensure that they are resistant against many kinds of cryptanalysis. Cryptanalysis is partly kind of academic fun, but there exists also a very real need for it. A flaw in a cryptosystem can cause big monetary and other losses to people using it. For example, assume that a big company encrypts its trade secrets using a weak cipher. Some competitor might know about the weakness and be able to learn about the trade secrets. This is an even worse situation than that the secrets were just stored without any encipherment at all. If they were not encrypted at all, the company owning them would know that outsiders can perhaps learn about them. However, now the weak cipher gives a misleading feeling of security.

Naturally, cryptographic *protocols* cannot be stronger than the cryptosystems they employ. However, security protocols can also contain flaws that are not a consequence of some cryptanalytic flaws in the used algorithms, *but are instead caused by the higher-level structure of the protocol itself.* Thus, such flaws are more or less present despite the choice of the cryptofunctions used to implement the protocol.

Nowadays, analyzing cryptographic protocols in order to find this kind of flaws — and to prevent them — has received a lot of attention. Quite surprisingly, a very large portion of proposed cryptographic protocols have been later found to be flawed. This has naturally encouraged research on this area. To give some examples, the Needham-Schröder symmetric key protocol [71] was shown flawed three years after its publication by Denning and Sacco [19]. They proposed a new public-key protocol for key distribution. Unfortunately, it was flawed also as shown by Abadi and Needham; they reported also other problems with some other protocols [1]. Gavin Lowe found the Needham-Schröder public-key protocol [71] to be flawed in 1995, almost twenty years after its invention [44]. Many other examples exist.

Some of the flaws in security protocols are purely logical mistakes. Such flaws are typically present regardless of how the protocol is actually implemented. Some other flaws are more tightly bound to the implementation details. As an example, we have presented some of the nontrivial bugs we found in an old version of the SSH-1 protocol [34]. This far, the formal analysis of cryptographic protocols has mostly focused on the purely logical flaws.

1.1.2 Threat Model

The main purpose of analyzing a security protocol is to examine the ways that malicious attackers can intervene with it and cause it to malfunction. The assumed computational abilities of the attackers and the relations between the attackers and the 'honest' protocol participants constitute together the *threat model.* The almost exclusively used threat model is the worst-case model that was first proposed, although in a less clear form, by Dolev and Yao [22]. This model is depicted in Figure 1.1. In this model, a set of honest participants communicates via an environment that is totally controlled by an attacker. Thus, the environment *is the* attacker. (Here 'honesty' means that a participant proceeds exactly according to the protocol being verified and does not divert from the specification at any moment.)

This model is very general. It is not necessary to assume the existence of

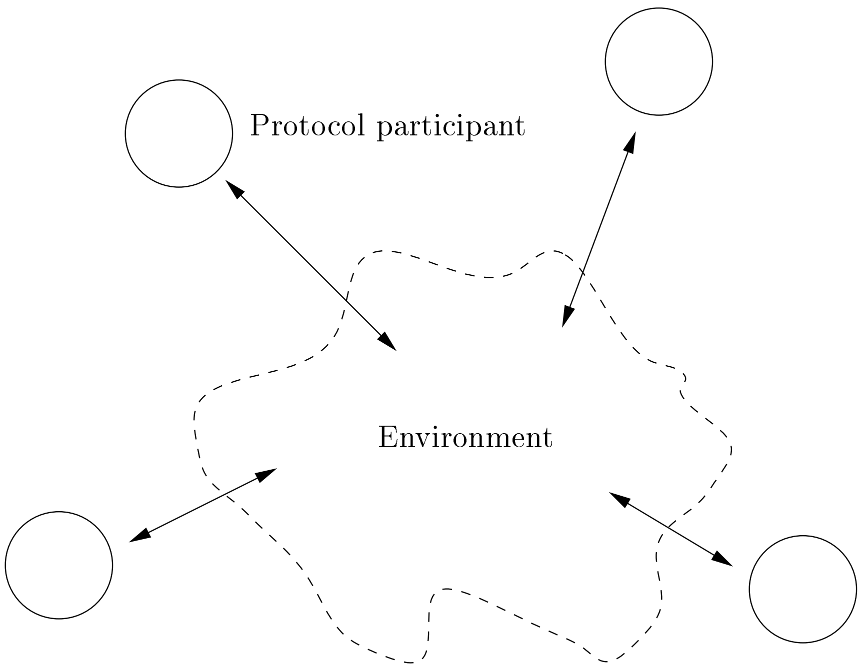

Figure 1.1 The worst-case threat model.

multiple attackers, because together they have no more abilities than the single omnipotent environment. Dishonest participants need not to be considered at all. They can be included in the environment. Also, usually it is not interesting to analyse situations where the attacker has less power than in the worst-case model, because security protocols should work correctly even in the pessimistic situation.

1.1.3 Typical Attacks

What kind of attacks do there exist against security protocols? This question cannot be answered before have defined what we *expect* from a given security protocol. As an example, assume that the purpose of a certain protocol is to distribute a session key safely between, say, Alice and Bob (this is a very common goal). Then we expect that after the protocol has been successfully terminated, there exists a session key that is known to Alice and Bob alone. If the protocol can terminate successfully so that an outsider knows the key then our expectations have not been realized. There is an *attack* against the protocol.

Attacks can be divided into active and passive attacks. Passive attacks are attacks that can be carried by listening to the communications between the honest participants, without modifying or redirecting them. All other attacks are active attacks. Typically almost all attacks that are mentioned in the literature are active. (Perhaps because passive attacks are somehow 'trivial' and thus are not so common in thoughtfully designed protocols.)

Let us now enumerate some typical attack classes. The classes we mentioned are in no way mutually disjoint. The purpose of this presentation is just to give an intuitive picture of the problems that can exist. **(1)** Two participants try to establish a secret session key. An attacker can lure them to a state where they think the protocol ended correctly. However, at the same time the attacker knows the secret session key. **(2)** An attacker can cheat Alice so that Alice thinks she is speaking with Bob. However, in the reality Alice is in connection with the attacker. **(3)** An attacker can replay some old message to a participant so that the participant thinks it is a new, fresh message. This kind of attacks are called *replay attacks*. **(4)** An attacker can use a honest participant to decrypt messages so that the attacker can read them. This despite the fact that the participant's intention is not to deliberately reveal the encrypted data to the attacker. This kind of attacks are often called *oracle attacks*.

1.1.4 Idealization

Typically, cryptographic protocols are analyzed in an idealized form. This means that most of the details of the actual cryptosystems are ignored. Ideal cryptography is assumed. Encryption, decryption and other operations are modeled using only the natural algebraic laws they always obey. All data that is exchanged in a cryptographic protocol is presented in symbolic form. For example, a simple protocol where the participant A sends a message to B and B returns it encrypted with the key k could be represented as:

$$\begin{array}{ccccc} A & \longrightarrow & B & : & m \\ B & \longrightarrow & A & : & \mathbf{e}(k,m) \end{array}$$

Here everything "irrelevant" has been abstracted away, such as the internal structure of the message m.

1.1.5 Analysis Methods

Analysis methods fall roughly into two classes. In the first class, doxastic logics — modal logics of belief — are used to analyse the evolution of the protocol participants' beliefs during the course of a protocol. The second class constitutes of methods that directly examine a protocol's state space.

1.1.6 Doxastic Logics

The seminal work on the use of doxastic logics for the analysis of cryptographic protocols is [12, 13]. The logic presented therein, the BAN logic, has inspired a large amount of variants and improvements [2, 15, 16, 23, 29, 32, 36, 40, 51, 82, 85, 86, 89, 91].

These logics usually lack good semantic foundation, and proofs are always done in axiomatic (Hilbert-)style. People have experimented with automatizing the proof process [10, 11, 41, 54, 55]. The logics are not free of problems and have only a limited domain of applicability [9, 14, 90, 93, 94]. As this work is about state space enumeration, doxastic logics are not considered more.

1.1.7 State Space Exploration

State space enumeration is a standard tool for the analysis of general concurrent systems. It is thus no wonder that it has been applied to the domain of security protocols as well. Some state space enumeration based methods analyse a finite state space, some others try to analyse an infinite one. The infiniteness of the state spaces of cryptographic protocols is caused by two factors.

The first one is that typically there is an infinite amount of different messages that can be sent by the environment to the participants. When generic concurrent systems are analyzed typically some of the data flow can be abstracted away. This not the case for security protocols. Instead data is of the essence.

The second reason is that some approaches try to analyse a system where the number of communicating parties is not restricted a priori. An unbounded number of participants naturally leads to an unbounded (infinite) state space. While the first kind of infinity can be coped with, the second kind causes the problem to become undecidable. A proof outline of this fact can be found in Appendix A.

Let us now look on some existing tools and methods. The NRL Protocol Analyzer [38, 56, 57, 59, 61, 83, 84] is a tool developed in CHACS. It performs backwards reachability analysis for a system consisting of an unbounded number of participants. Not surprisingly, the tool cannot always give a decisive answer. User interaction is sometimes required to guide the process and give hand-proofs of certain lemmas, although these proofs are nowadays also semi-automatized [60]. The NRL Protocol Analyzer is well known in the academic community because a lot has been written about it. However, it is not publicly available.

The Interrogator [38, 62, 63] is another tool for doing backwards analysis. Like the NRL analyzer it has also been written in PROLOG. Worth of mention is the Interrogator's ability to handle associative-commutative operators, such as exclusive-or, better than many of the other tools.

Gavin Lowe and Bill Roscoe used the CSP [33] model checker FDR [25, 77] for the analysis of cryptographic protocols, doing analysis in a finite state space [45, 46, 48, 50, 59, 78]. Notably, Gavin Lowe found a previously undiscovered flaw in the Needham-Schröder public-key protocol [44, 45]. It then became a new standard example of a faulty protocol. Lowe has also considered the problem of obtaining results for an unbounded number of participants [49] and has developed a compiler for translating high-level protocol descriptions into CSP formulae [47]. Steve Schneider has also examined the use of CSP [80].

John Mitchell et al applied the generic-purpose verification tool Murφ to the analysis of cryptographic protocols [65–67, 81]. Other generic tools have been also used, such as Astral [18] and LOTOS tools [7, 26]. Usually these approaches work in a finite state space. General information about the finite-state analysis of cryptographic protocols can be found e.g. in [64].

Lawrence Paulson used the theorem prover Isabelle to prove inductively properties of security protocols [4, 5, 73–76]. While this method does not explicitly enumerate state space, it is closer to the state space enumeration method than the use of doxastic logics.

There are many other methods that have been proposed for the analysis of security protocols. Many survey articles and good starting points exist in the literature [6, 17, 31, 38, 58, 79].

1.1.8 Design vs. Analysis

Developing analysis methods for cryptographic protocols that already exist is a sort of reactive act. Naturally, also proactive work has been made to guide the *design* of the security protocols in the first place [1, 3, 8, 27, 28, 30, 52, 53, 87, 88, 92]. This thesis does not, however, consider design issues.

1.2 Scope of This Thesis

This scope describes a new special-purpose tool for the analysis of cryptographic protocols. The tool works in the infinite state space of the first kind. The reason for confining to a bounded number of participants is the undecidability of the more general problem. This thesis describes the theory behind the tool and some of the implementation issues.

Chapter 2

Overview

An overview of the work is presented, including the theoretical model, the way security is defined and the implemented tool.

This thesis describes the development and implementation of a tool for analyzing cryptographic protocols. In this chapter, an overview of the work is presented.

Our method is the first method known to us that enumerates the complete infinite state space without encountering decidability problems nor requiring user interaction.

2.1 Theory

2.1.1 Symbolic State Space Search

The analysis method the tool employs to analyse cryptoprotocols is *symbolic state space search*. Symbolic state space search is a form of state space enumeration. Conventional state space enumeration focuses on individual states and enumerates them basically one by one. Symbolic search raises the level of abstraction. Instead of enumerating single states it handles state sets that are represented in some symbolic form. In other words, the sets are not represented merely as a collection of the individual states, but for example as a predicate that is true exactly for the states contained in the set. In this way also infinite sets can have finite representations.

2.1.2 Honest Participants

We model the honest participants of a protocol as automata that cannot execute arbitrarily long. As a matter of fact, the automata are represented as loopless programs. The programs constitute of individual actions of sending, receiving and generating messages, and conditionals.

There are good reasons for confining to loopless programs. Most cryptoprotocols can be represented by such programs. On the other hand, introducing nontrivial loop constructions, such as a generic while loop, would easily lead to undecidable analysis problems. The looplessness means also that all protocol runs are bounded *a priori* in length.

2.1.3 Model of Cryptography

The cryptographic primitives that the protocols constitute of are modeled using a rewrite system. Using rewrite systems for this task is quite common.

However, there is a twist. We would like to have the user to be able to define the rewrite system to use because that would give more flexibility to the method. On the other hand, the rewrite system must be *convergent* in order for the method to work. In our method there is a fixed set of rewrite rules that can be assigned to different user-defined operators. We also give a set of restrictions concerning the assignments. If the restrictions hold then the resulting system is guaranteed to be convergent.

The user is thus able to define the behavior of different cryptographic primitives within the bounds of the parameterized rewrite system. The important restriction here is that associative-commutative operations cannot be modeled. Thus, for example, the associative property of exclusive-or cannot be handled. While incorporating AC-operators would be interesting and quite possible, they were deliberately dropped from the scope of this work.

For the analysis, the tool employs the worst-case model depicted earlier and assumes ideal cryptography.

2.1.4 Security Assertions

When we analyse a system we basically want to find out whether or not a certain property holds for the system. For security protocols these properties capture some notion of security. In our work the security properties are presented using security assertions. Security assertions are predicates over the terminal states of a protocol's state space graph. The predicates can be composed of certain atomic formulas using standard boolean connectives and quantification over participants. The tool either verifies that the assertions hold in all terminal states or explicitly demonstrates states where the assertions do not hold.

2.2 Tool and its Capabilities

The tool is an end-user tool for the analysis of cryptographic protocols.

2.2.1 Input Language

The user describes a protocol by writing a text file that contains **(1)** definitions concerning the cryptosystems used in the protocol, **(2)** program code that describes how the protocol participants work, and **(3)** possibly some declarations changing the way the tool works. The programming language used to describe the participants' behavior is a simple procedural language that is designed to suit the particular purpose. The tool parses the input, analyses the protocol described and outputs the results of this analysis to the user.

The programming language reflects the formal model that is developed later in this thesis. Therefore, most importantly, it does not contain loop constructs.

2.2.2 Security Assertions in the Tool

Defining security is not easy, and the tool itself does not try to give any definition of security. Instead, the user who describes the protocol must also state what are the security assertions that must be checked. Security assertions are here logical formulas that can describe things such as 'the message X is not known to the environment' or 'the participant X and the participant Y have associated the same message to the context C'. Combinations of formulae can be formed by using standard logical connectives. There is also a possibility to use existential and universal quantifiers over participant names. The security assertions can be linked with a particular end state of a participant's protocol run — typically, a security assertion needs to hold only if a participant considers the protocol to have ended successfully.

An important restriction here is that the security assertions cannot describe conditions upon the relative execution order of distinct participants. Therefore it is not possible to require, for example, that two participants must have proceeded strictly in a lock-step fashion if the protocol has terminated successfully from the participants' point of view. There are two reasons for this restriction. The first is that when the relative execution order of the participants does not need to be considered there is potential for more state space reductions. The second is a matter of opinion. We see that proceeding in a certain order is not a security goal, and that it is better to express the required conditions directly as properties of the terminal states.

2.2.3 Implementation

The tool is implemented in Scheme [37], a semi-functional programming language that is a well-established dialect of LISP. The choice of Scheme was mainly motivated by the fact that most of the analysis process consists of manipulating complex symbolic expressions, and LISP suits well for symbolic computation. Also, Scheme is a high-level language and therefore programming effort was assumed beforehand to be smaller when compared e.g. to implementing the tool in C [39].

Part I

Theory

Chapter 3

Prerequisites

Prerequisites for fully understanding this thesis are given and mathematical notation is fixed. The covered areas are set theory, relations, orders, induction, terms, algebras and rewrite systems to the extent required in this work.

3.1 Basic Tools

3.1.1 Natural Numbers

The set $\mathbb{N}$ is the set of natural numbers, containing every nonnegative integer: $\mathbb{N} = \{0, 1, 2, \ldots\}$. The set of positive integers is denoted by $\mathbb{Z}_+ = \{1, 2, \ldots\}$.

3.1.2 Sets and Relations

A set is an unordered collection of objects. If the number of objects in the set S is finite the set can be written explicitly down as $\{a_1, \ldots, a_n\}$, where $a_1, \ldots, a_n$ are the objects contained in the set. The number of objects in S is said to be S's *cardinality*, denoted by $|S|$. For a finite S, $|S|$ belongs to $\mathbb{N}$. If $|S| = 0$ then S is the unique empty set $\emptyset$. The cardinality of $\mathbb{N}$ is $\aleph_0$, the first transfinite cardinal. Two infinite sets have the same cardinality iff there exists a bijection between the two sets. Every set that has the same cardinality as $\mathbb{N}$ is said to be *countably infinite*. Larger sets are *uncountable*.

If a is contained in S we write $a \in S$, otherwise $a \notin S$. A set can be specified as a set comprehension $\{x \mid c(x)\}$, denoting the set of all objects x for which the condition c holds. For example,

$$\{x \mid x \in \mathbb{Z}_+ \wedge x \equiv 0 \pmod 2\} \tag{3.1}$$

denotes the set of even positive integers.

The union of S and S' is $S \cup S' = \{x \mid x \in S \vee x \in S'\}$, their intersection $S \cap S' = \{x \mid x \in S \wedge x \in S'\}$ and their difference $S \setminus S' = \{x \mid x \in S \wedge x \notin S'\}$. A set S is a subset of S', denoted by $S \subseteq S'$ if $x \in S$ implies $x \in S'$. S is a proper subset of S', denoted by $S \subset S'$, if additionally $S \neq S'$. S and S' are equal iff $S \subseteq S'$ and $S' \subseteq S$.

The cartesian product of $S_1, \dots, S_n$ is denoted by $S_1 \times \cdots \times S_n$ and contains every ordered tuple $\langle x_1, \dots, x_n \rangle$ such that $x_k \in S_k$ for $k \in \{1, \dots, n\}$. An 1-tuple $\langle x \rangle$ is identified with the item x itself. Because tuples are ordered the sets $S \times T$ and $T \times S$ are different sets whenever $S \neq T$.

A relation on the sets $S_1, \dots, S_n$ is a subset of $S_1 \times \cdots \times S_n$. If R is a relation then we denote $\langle x_1, \dots, x_n \rangle \in R$ by $R(x_1, \dots, x_n)$. A binary relation is a relation on two sets. A relation on one set is a *predicate*. A binary relation R on A is **(1)** reflexive if always $R(a, a)$; **(2)** symmetric if always $R(a, b) \Rightarrow R(b, a)$; **(3)** transitive if always $R(a, b) \wedge R(b, c) \Rightarrow R(a, c)$; **(4)** antisymmetric if always $R(a, b) \wedge R(b, a) \Rightarrow a = b$; and **(5)** total if either $R(a, b)$ or $R(b, a)$ whenever $a \neq b$.

Any reflexive and transitive relation is a *quasi-ordering*. Antisymmetric quasi-orderings are *partial orderings*. Total partial orderings are *total orderings*. Any reflexive, transitive and symmetric relation is an *equivalence relation*. A binary relation $\prec$ is *well-founded* if there does not exist an infinite descending chain $\cdots \prec a_i \cdots \prec a_1 \prec a_0$. If $\sqsubseteq$ is an ordering over P, then $\langle P, \sqsubseteq \rangle$ is the corresponding *order*. For example $\langle \mathbb{N}, \leq \rangle$ is a well-founded total order.

For any binary relation $\rightarrow$, $\rightarrow^*$ denotes the transitive and reflexive closure of $\rightarrow$, which is the smallest transitive and reflexive relation containing $\rightarrow$. $\rightarrow^+$ denotes the transitive closure, which is the smallest relation containing $\rightarrow$ and being transitive.

A function is a binary relation R such that

$$R(a, b) \wedge R(a, b') \Rightarrow b = b'. \tag{3.2}$$

If f is a function, we use the notation $f(x_1, \dots, x_k) = y$ to denote that $\langle \langle x_1, \dots, x_k \rangle, y \rangle \in f$. In this case, the value of f on $x_1, \dots, x_k$ is said to be y. If $\nexists y : \langle \langle x_1, \dots, x_k \rangle, y \rangle \in f$, we denote $f(x_1, \dots, x_k) = \perp$. In this case, the value of f on $x_1, \dots, x_k$ is said to be undefined.

We denote by $f : A \mapsto B$ the fact that $f \subseteq A \times B$. A is the *domain* of f and B its *range*. If $f(a) \neq \bot$ for all $a \in A$, f is said to be *total*. A *partial function* is a function that is not necessarily total.

The set of functions from A to B is denoted by B^A. Specially, the set of all the subsets of A is denoted by 2^A (motivated by the fact that a subset of A can be represented as a function $f : A \mapsto \{0, 1\}$). This is the *powerset* of A.

Given a function f, we denote by $f[a \mapsto b]$ the new function that is defined as

$$f[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise.} \end{cases} \tag{3.3}$$

If f and g are functions $f : A \mapsto B$ and $g : B \mapsto C$ then $g \circ f$ is the function $g \circ f : A \mapsto C$ defined as

$$(g \circ f)(x) = g(f(x)). \tag{3.4}$$

If $f : A \mapsto B_1 \times \cdots \times B_n$ is a function, we denote by f_i the new function that is defined by

$$f(a) = \langle x_1, \ldots, x_n \rangle \Rightarrow f_i(a) = x_i. \tag{3.5}$$

f_i is said to be the *ith projection of f*.

A *closure operator* C on a set S is a function $C : 2^S \mapsto 2^S$ such that, for any $A, B \subseteq S$, (1) $A \subseteq C(A)$, (2) $C(C(A)) = C(A)$, and (3) $A \subseteq B \Rightarrow C(A) \subseteq C(B)$.

3.1.3 Induction

The *principle of well-founded (noetherian) induction*[1] states that, for any predicate P on a set A and a well-founded relation $\prec$ over A,

$$(\forall a \in A)[P(a)] \quad \text{iff} \quad (\forall a \in A)[(\forall b \prec a)[P(b)] \Rightarrow P(a)]. \tag{3.6}$$

The *principle of mathematical induction* is the specialization of well-founded induction to the case where $A = \mathbb{N}$ and $a \prec b \Leftrightarrow b = a + 1$. Another

[1] This principle can be proven as follows: ($\Rightarrow$) Assume, for the sake of contradiction, that $\neg P(a)$ for some $a \in A$. Then there must be a minimal element m in $\{a \in A \mid \neg P(a)\}$ because $\prec$ is well-founded. But then $\neg P(m)$ and yet $(\forall b \prec m)[P(b)]$, which contradicts the assumption. ($\Leftarrow$) Trivial.

special case is the case $A = \mathbb{N}$ and $a \prec b \Leftrightarrow a < b$ which is called *course-of-values* induction. *Structural induction* is well-founded induction where A is a collection of some hierarchical structures and $\prec$ represents a substructure relation.

3.2 Terms, Algebras and Related Things

3.2.1 Terms

Let $\mathscr{F}_0, \mathscr{F}_1, \ldots$ be collections of symbols. Let $\mathcal{S} = \bigcup_{n \in \mathbb{N}} \mathscr{F}_n$, *a signature*, be given. Let Q be any set. We denote by $\mathbf{T}_\mathcal{S}(Q)$ the smallest set satisfying

1. $x \in Q \Rightarrow x \in \mathbf{T}_\mathcal{S}(Q)$, and
2. $x_1, \ldots, x_n \in \mathbf{T}_\mathcal{S}(Q), f \in \mathscr{F}_n \Rightarrow f(x_1, \ldots, x_n) \in \mathbf{T}_\mathcal{S}(Q)$.

This is the set of $\mathcal{S}, Q$-*terms.*

Observe that there can be nullary operators in the signature, i.e. $\mathscr{F}_0$ can be nonempty. Therefore two kinds of 'constants' can appear in an $\mathcal{S}, Q$-term: those that belong to $\mathscr{F}_0$ and those that belong to Q. This technical complication is introduced to make later developments more streamlined.

A *congruence relation* over $\mathbf{T}_\mathcal{S}(Q)$ is an equivalence relation $\equiv$ over $\mathbf{T}_\mathcal{S}(Q)$ for which the implication

$$t_1 \equiv s_1, \ldots, t_k \equiv s_k \Rightarrow f(t_1, \ldots, t_k) \equiv f(s_1, \ldots, s_k) \tag{3.7}$$

holds universally.

Let $\mathcal{S}$ be a signature as before and $\mathcal{X}$ be a countable set of variables such that $\mathcal{S} \cap \mathcal{X} = \emptyset$. A *substitution* σ is a function $\mathcal{X} \mapsto A$ for some set A such that $\sigma(x) \neq x$ only for finitely many x. If the set of variables not mapped to themselves is $\{x_1, \ldots, x_n\}$, the corresponding substitution can be written as $\{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$. The set of all substitutions $\sigma : \mathcal{X} \mapsto A$ is denoted by $\mathbf{subst}(\mathcal{X}, A)$.

The domain of a substitution $\sigma \in \mathbf{subst}(\mathcal{X}, A)$ can be implicitly extended to $\mathbf{T}_\mathcal{S}(\mathcal{X})$ by defining

$$\sigma(f(t_1, \ldots, t_k)) = f(\sigma(t_1), \ldots, \sigma(t_k)). \tag{3.8}$$

This extension will be automatically assumed in the rest of this work.[2]

If $\mathbf{T}_{\mathcal{S}}(\mathcal{X})$ is a collection of terms with variables, then $\mathbf{T}_{\mathcal{S}}(\emptyset) \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ is the set of *ground terms*. From now on, the phrase "ground term" refers explicitly to an element of $\mathbf{T}_{\mathcal{S}}(\emptyset)$, i.e. to the $\mathcal{S}, \emptyset$-terms. A "term" can be either an element of $\mathbf{T}_{\mathcal{S}}(\emptyset)$ or of $\mathbf{T}_{\mathcal{S}}(\mathcal{X})$ or something else; any particular meaning should be clear from the particular context.

A *ground substitution* with respect to $X \subseteq \mathcal{X}$ is a substitution whose range belongs to $\mathbf{T}_{\mathcal{S}}(\emptyset)$ when its domain is restricted to X. When X can be understood it can be omitted. In this case, a ground substitution σ is grounding for all "interesting" variables.

A term t *matches* another term s if there exists a substitution σ such that $\sigma(s) = t$; in this case s is said to *subsume* t, which is denoted by $s \trianglelefteq t$. The relation $\trianglelefteq$ is a quasi-ordering on terms. In the same vein, a substitution σ subsumes σ' if there exists a substitution σ'' such that $\sigma'' \circ \sigma = \sigma'$. This is also denoted by $\sigma \trianglelefteq \sigma'$, and σ is said to be at least as general as σ'. If also $\sigma' \not\trianglelefteq \sigma$, σ is said to be more general than σ'; this is denoted by $\sigma \triangleleft \sigma'$.

For two terms t and s, a *unifier* is a substitution σ such that $\sigma(t) = \sigma(s)$. If σ is a unifier of t and s and for all unifiers σ' of t and s, $\sigma \trianglelefteq \sigma'$, σ is said to be the *most general unifier* (mgu) of t and s (the mgu is unique when it exists). The process of computing the mgu of two terms is called *unification*.

We define the function

$$\mathrm{unify}(t, s) = \begin{cases} \text{mgu of } t \text{ and } s & \text{when it exists} \\ \perp & \text{otherwise.} \end{cases} \tag{3.9}$$

The definition of unify assumes that $\mathcal{X}$ and $\mathcal{S}$ are understood.

3.2.2 Algebras

An $\mathcal{S}$-algebra $\mathscr{A}$ consists of a *carrier set* A and a set of functions $\mathscr{F}_{\mathscr{A}}$ such that there is a correspondence θ between $\mathcal{S}$ and $\mathscr{F}_{\mathscr{A}}$ so that if $f \in \mathscr{F}_n$, then $\theta(f, n) = f_{\mathscr{A}}$ and $f_{\mathscr{A}} : A^n \mapsto A$. Later, we deliberately abuse notation and denote the carrier set of an algebra $\mathscr{A}$ by $\mathscr{A}$ itself.

An $\mathcal{S}$-algebra attaches a unique meaning to every element of $\mathbf{T}_{\mathcal{S}}(\emptyset)$. This notion is made explicit by the introduction of the semantical function $[\![\cdot]\!]_{\mathscr{A}}$,

[2] Note that a substitution corresponds to the idea of *simultaneously* replacing the occurrences of substituted variables with their substituents. If $\sigma = \{x_1 \mapsto x_2, x_2 \mapsto x_3\}$, then $\sigma(f(x_1, x_2)) = f(x_2, x_3)$ and not, say, $f(x_3, x_3)$.

defined as follows:

$$[\![f(x_1, \ldots, x_n)]\!]_{\mathscr{A}} = f_{\mathscr{A}}([\![x_1]\!]_{\mathscr{A}}, \ldots, [\![x_n]\!]_{\mathscr{A}}). \tag{3.10}$$

For every signature $\mathcal{S}$, there exists a "trivial" algebra $\mathbf{F}(\mathcal{S})$ whose carrier set is $\mathbf{T}_{\mathcal{S}}(\emptyset)$ and for which

$$[\![x]\!]_{\mathbf{F}(\mathcal{S})} = x. \tag{3.11}$$

This algebra is unique for any $\mathcal{S}$ and is called the *free term algebra over $\mathcal{S}$*.

Let $\mathscr{A}$ and $\mathscr{B}$ be two $\mathcal{S}$-algebras. A mapping $\varphi : \mathscr{A} \mapsto \mathscr{B}$ satisfying

$$\varphi(f_{\mathscr{A}}(x_1, \ldots, x_k)) = f_{\mathscr{B}}(\varphi(x_1), \ldots, \varphi(x_k)) \tag{3.12}$$

for all $f \in \mathscr{F}_k$ and $x_i \in \mathscr{A}$ is a *homomorphism* from $\mathscr{A}_1$ to $\mathscr{A}_2$. If φ exists, $\mathscr{A}_1$ is said to be *homomorphic* to $\mathscr{A}_2$. If $\mathscr{A}_2$ is also homomorphic to $\mathscr{A}_1$, the two algebras are said to be *isomorphic*.

If $\mathscr{A}$ is an algebra and $S \subseteq \mathscr{A}$ is a subset of the carrier set, the *algebraic closure* of S, denoted by $S^{*[\mathscr{A}]}$ is the smallest set S' such that (1) $S \subseteq S'$ and (2) if $f_{\mathscr{A}} \in \mathscr{F}_{\mathscr{A}}$ and $s_1, \ldots, s_n \in S'$ then also $f_{\mathscr{A}}(s_1, \ldots, s_n) \in S'$, provided that $f \in \mathscr{F}_n$. When the algebra is understood, the annotation $^{[\mathscr{A}]}$ can be omitted.

3.2.3 Rewrite Systems

A more thorough introduction to rewrite systems can be found in [20].

Any term $t \in \mathbf{T}_{\mathcal{S}}(\emptyset)$ can be seen as a finite ordered tree. The Dewey decimal notation gives a unique name for every position of a subterm in t. The notation is a sequence of natural numbers, describing the "path" to the subterm starting from the outermost term. For example, the subterm in position 2.3 in

$$t = f(a, g(c, d, e), b, c) \tag{3.13}$$

is e, i.e. the third subterm of t's second subterm. Thus positions are not terms but can denote them. The subterm referred to by a position p is denoted by $t|_p$ and called an occurrence.

By $t[s]_p$ we denote the term that is obtained by replacing in t the subterm denoted by p by the new subterm s. For example, $f(a, b, c)[d]_2 = f(a, d, c)$.

Let $\mathcal{S}$ be fixed and $\mathcal{X}$ a set of variables. A rewrite system R over $\mathbf{T}_{\mathcal{S}}(\emptyset)$ is a collection of directed equations, *rules*, over $\mathbf{T}_{\mathcal{S}}(\mathcal{X})$. The equations define the *rewrite relation* $\underset{R}{\rightarrow}$ over terms $\mathbf{T}_{\mathcal{S}}(\emptyset)$. $t \underset{R}{\rightarrow} s$ holds iff there exists a position p, a ground substitution σ and a rule $l \rightarrow r$ in R such that $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$.

Example 3.1 Let R be

$$f(X,Y) \rightarrow g(Y,X), \quad g(a,X) \rightarrow X. \tag{3.14}$$

where $X, Y \in \mathcal{X}$ and $a \in \mathbf{T}_{\mathcal{S}}(\emptyset)$. Then $f(b,a) \rightarrow_R g(a,b) \rightarrow_R b$. On the other hand, there is no t such that $g(b,c) \underset{R}{\rightarrow} t$. $g(b,c)$ is said to be *irreducible* and its own *normal form*. □

A normal form of the term t is any term s such that $t \underset{R}{\rightarrow}^* s$ and s is irreducible. If there exists exactly one normal form it is said to be *unique*. If every term has a unique normal form with respect to R then R is said to be *uniquely normalizing*. If in R there are no infinite derivations $t \underset{R}{\rightarrow} s \underset{R}{\rightarrow} \cdots$ the system is said to be *noetherian*. A noetherian and uniquely normalizing rewrite system is said to be *convergent* because every term 'converges' after a finite amount of applications of the rewrite system upon it to a standard, irreducible form.

Chapter 4

Protocols and Their Analysis

The model of cryptography is defined via a parameterized rewrite system. Protocol semantics are defined and the general analysis problem stated.

In this chapter, we define a particular model of cryptographic protocols.

In order to get a working model for security protocols we need to take stand at least on the following issues:

1. How is cryptography modeled?
2. How do the single participants of a protocol work? How are their workings described?
3. What is the threat model like?
4. What does security mean? How are the goals of a protocol expressed?

We are going to use the worst-case model presented in the introduction as our threat model. Thus there is one attacker that is in full control of the environment. Cryptography will be modeled using an algebra that is defined with the help of a rewrite system. In essence, the algebraic properties of cryptographic primitives will be modeled as rewrite rules.

We define a simple language that denotes loopless programs. Those programs will be used to express the behavior of the honest participants.

We give no specific notion of security. Instead, we define a language of predicates that can be used to express security assertions that must hold when a protocol has terminated. These security assertions are what the tool then verifies, given particular cryptographic primitives and particular behaviors of protocol participants.

4.1 Model of Cryptography

All cryptographic 'primitives', such as DES encryption, SHA hashing [69, 70], digitally signing a message etc., can be seen to form a particular algebra $\mathscr{C}$ over all binary strings. Unfortunately, this real-world algebra is not suitable for formal analysis because of its overwhelming computational complexity. If we were to work on this level of detail we would need to fully cryptanalyze all primitives used in a protocol and all their interactions before we could reach any result. For this reason, this algebra needs to be 'simplified'.

In the formal world, we simplify $\mathscr{C}$ by using another algebra that will be homomorphic to it.

The essential property of algebra homomorphisms is that if $\mathscr{A}$ is homomorphic to $\mathscr{B}$, then two terms that are equal under $\mathscr{A}$ are also (actually, their images) equal under $\mathscr{B}$, but not necessarily vice-versa.[1] Thus, $\mathscr{A}$ can be seen as an approximation of $\mathscr{B}$, capturing some of its properties but not all of them. (This depends on the point of view. On the other hand, $\mathscr{B}$ can be seen as an approximation of $\mathscr{A}$ because some terms that have distinct values under $\mathscr{A}$ are 'coalesced' to the same value under $\mathscr{B}$ and thus some information can be seen to be lost.)

Definition 4.1 Let a signature $\mathcal{S}$ and a convergent rewrite system $\mathfrak{R}$ over $\mathbf{T}_{\mathcal{S}}(\emptyset)$ be fixed. We denote by $\mathfrak{N}(t)$ the normal form of $t \in \mathbf{T}_{\mathcal{S}}(\emptyset)$, which exists because $\mathfrak{R}$ is convergent. □

Definition 4.2 Let a signature $\mathcal{S}$ and a convergent rewrite system $\mathfrak{R}$ over

[1] A homomorphism φ from $\mathscr{A}$ to $\mathscr{B}$ (two $\mathcal{S}$-algebras) is in essence a function satisfying

$$\varphi(\llbracket t \rrbracket_{\mathscr{A}}) = \llbracket t \rrbracket_{\mathscr{B}}. \tag{4.1}$$

Thus, if t and t' are $\mathcal{S}$-terms such that $\llbracket t \rrbracket_{\mathscr{A}} = \llbracket t' \rrbracket_{\mathscr{A}}$ we get the commutative diagram below, implying that t and t' are also equivalent under $\mathscr{B}$:

$$\begin{tikzcd} \llbracket t \rrbracket_{\mathscr{A}} \arrow[r, equal] \arrow[d, "\varphi"] & \llbracket t' \rrbracket_{\mathscr{A}} \arrow[d, "\varphi"] \\ \llbracket t \rrbracket_{\mathscr{B}} \arrow[r, equal] & \llbracket t' \rrbracket_{\mathscr{B}} \end{tikzcd} \tag{4.2}$$

$\mathbf{T}_{\mathcal{S}}(\emptyset)$ be given. We denote by $\mathscr{M}(\mathcal{S}, \mathfrak{R})$ the $\mathcal{S}$-algebra that is defined as

$$[\![x]\!]_{\mathscr{M}(\mathcal{S},\mathfrak{R})} = \mathfrak{N}(x). \tag{4.3}$$

The carrier set of $\mathscr{M}(\mathcal{S}, \mathfrak{R})$ is naturally the set of terms that are in their normal forms. □

We will use $\mathscr{M}(\mathcal{S}, \mathfrak{R})$-algebras to model the real-world cryptoalgebras. Observe that $\mathscr{M}(\mathcal{S}, \mathfrak{R})$ is isomorphic to the free term algebra $\mathbf{F}(\mathcal{S})$ modulo the congruence $\equiv$ defined as

$$t \equiv s \quad \text{iff} \quad \mathfrak{N}(t) = \mathfrak{N}(s). \tag{4.4}$$

Many properties of cryptographic functions are such that they can be easily expressed as rewrite rules. For example, symmetric encryption can be expressed with the rule

$$d(K, e(K, X)) \to X \tag{4.5}$$

if d and e are the corresponding decryption and encryption operations. Thus e.g. $d(a, e(a, c))$ and c would be congruent under $\equiv$, formalising the idea that decryption cancels encryption. Of course, in order for $\mathscr{M}(\mathcal{S}, \mathfrak{R})$ to be a good model of cryptography, $\mathfrak{R}$ must be chosen thoughtfully.

4.1.1 Parametrised Rewrite System $\mathfrak{R}$

There exist different cryptosystems with different properties. Therefore it is not a good idea to completely fix the rewrite system $\mathfrak{R}$. Instead, we adopt the following approach.

Definition 4.3 Let $\mathcal{S}$ be a given signature. Let $\mathscr{D}$ be a finite set containing zero or more of the following items:

1. $P_{k\text{-tuple}}(z, s_1, \ldots, s_k)$, when $s_1, \ldots, s_k \in \mathscr{F}_1$ and $z \in \mathscr{F}_k$;
2. $P_{\text{symenc}}(e, d)$, when $e, d \in \mathscr{F}_2$;
3. $P_{\text{pubenc}}(e, d, p, s)$, when $e, d \in \mathscr{F}_2$ and $p, s \in \mathscr{F}_1$;
4. $P_{\text{signs}}(c, v, p, s)$, when $c \in \mathscr{F}_2$, $v \in \mathscr{F}_3$ and $p, s \in \mathscr{F}_1$; and
5. $P_{\text{inv}}(a, b)$, when $a, b \in \mathscr{F}_1$.

Then $\mathscr{D}$ is an *operator specification.* □

Definition 4.4 Let $\mathcal{S}$ be a given signature such that TRUE $\in \mathscr{F}_0$ and $\mathscr{D}$ a corresponding operator specification. We define $\mathfrak{R}(\mathscr{D})$ to be the rewrite system obtained by instantiating every rule in Figure 4.1 for every set of operators such that the corresponding instance of the operator definition — shown to the left of the rule — belongs to $\mathscr{D}$.[2] □

Example 4.1 If $\mathscr{D} = \{P_{\text{inv}}(f, g), P_{\text{2-tuple}}(a, b, c)\}$, then $\mathfrak{R}(\mathscr{D})$ would be

$$f(g(M)) \to M, \quad b(a(A_1, A_2)) \to A_1, \quad c(a(A_1, A_2)) \to A_2. \square \tag{4.6}$$

Intuitively, $P_{\text{symenc}}(e, d)$ specifies that e and d are symmetric encryption and decryption operators, respectively. $P_{\text{pubenc}}(e, d, p, s)$ defines a public-key cryptosystem such that $p(k)$ is the public key and $s(k)$ the private key derived from the common secret key material k, and that e is the encryption operator and d the decryption operator. $P_{\text{signs}}(c, v, p, s)$ defines a signature system such that v is used to verify signatures that are created using c and the secret key $s(k)$. Verification with v requires the knowledge of the signed message and the corresponding public key $p(k)$. $P_{\text{inv}}(a, b)$ defines a as the inverse of b. Finally, $P_{k\text{-tuple}}(c, s_1, \ldots, s_k)$ defines that c creates compound structures of k submessages such that the submessages can be accessed by the selectors $s_1, \ldots, s_k$.

Example 4.2 With a suitable choice of $\mathscr{D}$, $\mathfrak{R}(\mathscr{D})$ can model many different cryptographic functions. Here are some examples:

DES-encryption $\{P_{\text{symenc}}(\mathsf{DESenc}, \mathsf{DESdec}), P_{\text{symenc}}(\mathsf{DESdec}, \mathsf{DESenc})\} \subseteq \mathscr{D}$. These two specifications together model a generic block cipher where encryption is canceled by decryption but also decryption by encryption.

RSA-encryption $\{P_{\text{pubenc}}(\mathsf{RSAenc}, \mathsf{RSAenc}, \mathsf{RSApubk}, \mathsf{RSAprivk})$, $P_{\text{pubenc}}(\mathsf{RSAenc}, \mathsf{RSAenc}, \mathsf{RSAprivk}, \mathsf{RSApubk})$, $P_{\text{inv}}(\mathsf{RSAkeym}, \mathsf{RSAprivk})\} \subseteq \mathscr{D}$. In RSA, encryption and decryption are the same operation but with different keys. Also, decryption can be canceled by encryption, thus the first two specifications are needed. The last specification gives RSAprivk an inverse, RSAkeym. The purpose of RSAkeym is to enable the intruder to obtain the secret key

[2] The set of available rewrite rules is redundant in two ways. First, 1-tuples correspond to invertible functions, i.e. the rule (4.11) could be removed. The reason for including the invertible function rule is mostly historical. The second form of redundancy is in that symmetric encryption could be (almost) modeled by public-key cryptography, i.e. the ruls (4.8) could be removed. However, it is conceivable that handling the rule (4.8) will be computationally cheaper than handling the rule (4.9). Therefore, using (4.8) where possible should lead to more efficient analysis than implementing the semantics of (4.8) via (4.9).

$$P_{k\text{-tuple}}(c, s_1, \ldots, s_k) \Rightarrow \qquad s_i(c(A_1, \ldots, A_k)) \to A_i \tag{4.7}$$

$$P_{\text{symenc}}(e, d) \Rightarrow \qquad d(K, e(K, M)) \to M \tag{4.8}$$

$$P_{\text{pubenc}}(e, d, p, s) \Rightarrow \qquad d(s(K), e(p(K), M)) \to M \tag{4.9}$$

$$P_{\text{signs}}(c, v, p, s) \Rightarrow \qquad v(M, p(K), c(s(K), M)) \to \textsc{true} \tag{4.10}$$

$$P_{\text{inv}}(a, b) \Rightarrow \qquad a(b(M)) \to M \tag{4.11}$$

Figure 4.1 The rewrite rules $\mathfrak{R}$ used.

material if it gets a private key into its possession and therefore also enable it to obtain the public key.

RC4-encryption $\{P_{\text{symenc}}(\mathsf{RC4enc}, \mathsf{RC4enc})\} \subseteq \mathscr{D}$. Because RC4 is a stream cipher, decryption and encryption are identical operations.

Arithmetic incrementation $\{P_{\text{inv}}(1+, 1-), P_{\text{inv}}(1-, 1+)\} \subseteq \mathscr{D}$.

ElGamal-signature $\{P_{\text{signs}}(\mathsf{EGsign}, \mathsf{EGcheck}, \mathsf{EGpubk}, \mathsf{EGprivk})\} \subseteq \mathscr{D}$. With this specification, a signature formed by EGsign can be checked with EGcheck, given that a public key corresponding to the private key in the signature is available. ElGamal can be also used for encryption, but here encryption and decryption operators are different and decryption cannot be generally canceled by encryption, because encrypted messages are in different format than plaintext messages. Thus we would get $\{P_{\text{pubenc}}(\mathsf{EGenc}, \mathsf{EGdec}, \mathsf{EGpubk}, \mathsf{EGprivk})\} \subseteq \mathscr{D}$. □

4.1.2 Convergence of $\mathfrak{R}$

When defining $\mathscr{M}(\mathcal{S}, \mathfrak{R})$ we assumed that $\mathfrak{R}$ is convergent.[3] We want to set $\mathfrak{R} = \mathfrak{R}(\mathscr{D})$. However, it is not clear that $\mathfrak{R}(\mathscr{D})$ is convergent for an arbitrary $\mathscr{D}$. As a matter of fact, it is not. In order to ensure that $\mathfrak{R}$ is convergent the choice of $\mathscr{D}$ must be restricted. We do this by giving a set of restrictions that

[3] That $\mathfrak{R}$ is convergent corresponds to the following formal idea: If φ is the homomorphism from $\mathscr{M}$ to $\mathscr{C}$, then if the homomorphic images of m and m' are equivalent, then m and m' must have the same normal form under $\mathscr{M}$. In essence, every element of $\mathscr{C}$ — every binary string — can be represented by a unique term of $\mathscr{M}$. This is exactly the essence of our approximation of the real-world cryptography.

a suitable $\mathscr{D}$ must fulfill. Some of these restrictions are necessary for making $\mathfrak{R}$ convergent, but others could perhaps be replaced with weaker ones. This is not, however, very important because all the restrictions make sense in our context of security protocol analysis.

Consistent inverses (CI) *If $P_{inv}(a, b)$, $P_{inv}(b, c) \in \mathscr{D}$ then $c = a$.*

(If this does not hold then e.g. the term $(a(b(c(\text{TRUE}))))$ has the two normal forms $a(\text{TRUE})$ and $c(\text{TRUE})$.)

Intuitively, if f is the inverse of g and g is the inverse of h then actually h and f must be the same function.

Consistent symmetric systems (CSS) *If $P_{symenc}(a, b)$, $P_{symenc}(b, c) \in \mathscr{D}$ then $a = c$.*

(If this does not hold, then e.g. the term $c(k, b(k, a(k, \text{TRUE})))$ has the two normal forms $a(k, \text{TRUE})$ and $c(k, \text{TRUE})$.)

This is similar to (CI).

Consistent asymmetric systems (CAS) *If $P_{pubenc}(e, d, p, s)$, $P_{pubenc}(d', e, s', p) \in \mathscr{D}$ then $d = d'$ and $s = s'$.*

(If this does not hold, then e.g. the term $d(s(k), e(p(k), d'(s'(k), \text{TRUE})))$ has the two normal forms $d(s(k), \text{TRUE})$ and $d'(s'(k), \text{TRUE})$.)

This is similar to (CI).

Selectors not invertible (SNI) *If $P_{n\text{-}tuple}(c, a_1, \ldots, a_n)$, $P_{inv}(b, d) \in \mathscr{D}$ then $d \notin \{a_1, \ldots, a_n\}$.*

(If this does not hold but, say, $x = a_1$ then e.g. the term $b(d(c(1, \ldots, n)))$ has the two normal forms $c(1, \ldots, n)$ and $b(1)$.)

This restriction corresponds to the fact that typically selecting a part of a structure is not an invertible process because other parts of the structure are irrecoverably lost.

Constructors do not reduce (CDNR) *If $P_{n\text{-}tuple}(c, a_1, \ldots, a_n) \in \mathscr{D}$, then for any $x_1, \ldots,$ $P_{inv}(c, x_1)$, $P_{symenc}(x_1, c)$, $P_{pubenc}(x_1, c, x_2, x_3)$, $P_{signs}(x_1, c, x_2, x_3)$, $P_{n\text{-}tuple}(c', x_1, \ldots, c, \ldots, x_n) \notin \mathscr{D}$.*

(If this does not hold but, say, $P_{\text{2-tuple}}(c, a_1, a_2), P_{\text{symenc}}(e, c) \in \mathscr{D}$, then the term $a_1(c(1, (e(1, 2))))$ has the two normal forms $a_1(2)$ and 1.)

That constructors do not 'reduce' formalizes the thought that creating a collection of objects does not transform the collection in any non-trivial, irrecoverable way.

Matching selectors: (MS) *If* $P_{n\text{-}tuple}(c, a_1, \ldots, a_n)$, $P_{k\text{-}tuple}(c, b_1, \ldots, b_k) \in \mathscr{D}$ *and* $a_i = b_j$ *then* $i = j$.

(If this does not hold but, say, $P_{\text{2-tuple}}(c, a, b), P_{\text{2-tuple}}(c, b, a) \in \mathscr{D}$ then the term $a(c(1, 2))$ has the two normal forms 1 and 2.)

This restriction is very easy to justify; a selector should not refer ambiguously to multiple items of a compound structure.

Symmetric and asymmetric schemes distinct (SAASD) *If* $P_{symenc}(e, d)$, $P_{pubenc}(e', d', p, s) \in \mathscr{D}$ *then* $\{e, d\} \cap \{e', d'\} = \emptyset$.

(If this does not hold, then e.g. the following can happen: if $P_{\text{symenc}}(d, e), P_{\text{pubenc}}(e, d, p, s) \in \mathscr{D}$, then the term $e(s(k), d(s(k), e(p(k), \text{TRUE})))$ has the two normal forms $e(p(k), \text{TRUE})$ and $e(s(k), \text{TRUE})$.)

Intuitively, a symmetric cryptosystem cannot be an asymmetric one at the same time, and vice-versa. (Actually, this intuition is false because any symmetric scheme can be seen as an asymmetric scheme for which $p = s$. This is not important, though, so we use this 'too strong' requirement here.)

Stable key derivation functions (SKDF) *If* $P_{pubenc}(e, d, p, s)$, $P_{inv}(a, b) \in \mathscr{D}$, *then* $p, s \neq a$. *Similarly, if* $P_{1\text{-}tuple}(a, b) \in \mathscr{D}$ *then* $p, s \neq b$. *And, if* $P_{signs}(c, v, p, s), P_{inv}(a, b) \in \mathscr{D}$, *then* $p, s \neq a$; *similarly if* $P_{1\text{-}tuple}(a, b) \in \mathscr{D}$ *then* $p, s \neq b$.

(If this does not hold but e.g. $P_{\text{pubenc}}(e, d, p, s), P_{\text{inv}}(p, b) \in D$, then the term $d(s(b(k)), e(p(b(k)), \text{TRUE}))$ has the two normal forms $d(s(b(k)), e(k, \text{TRUE}))$ and TRUE.)

This restriction refers to the assumption that a key is a key despite of the material it has been derived of.

Signature not decryption (SND) *If* $P_{signs}(c, v, p, s)$, $P_{symenc}(a, b) \in \mathscr{D}$ *then* $c \neq b$. *Similarly, if* $P_{signs}(c, v, p, s), P_{pubenc}(e, d, p', s') \in \mathscr{D}$ *then* $c \neq d$.

(If this does not hold but e.g. $P_{\text{signs}}(c, v, p, s), P_{\text{symenc}}(a, c) \in \mathscr{D}$ then the term $v(a(s(k), m), p(k), c(s(k), a(s(k), m)))$ has the two normal forms TRUE and $v(a(s(k), m), p(k), m)$.)

At the first moment this seems to be somehow a contradictory restriction, because many digital signature schemes indeed utilize asymmetric decryption as the signing function. However, such signature schemes can be modeled using P_{pubenc}. P_{signs} is used to model signature schemes where this is *not* the fact, i.e. where the signature cannot be reverted to reveal the original message even if the secret key is known.

$$\begin{tikzcd} a \arrow[r, "\mathfrak{R}"'] \arrow[d, "\mathfrak{R}"'] & b \arrow[d, "\mathfrak{R}"', "*"] \\ c \arrow[r, "*", "\mathfrak{R}"'] & d \end{tikzcd} \tag{4.14}$$

Figure 4.2 Local confluence as a commutative diagram.

Definition 4.5 Let $\mathcal{S}$ be a given signature and $\mathscr{D}$ a corresponding operator specification. If $\mathscr{D}$ fulfils the requirements stated above, $\mathscr{D}$ is said to be *safe*. □

The next theorem shows that the restrictions above are sufficient for making $\mathfrak{R}$ convergent.

Theorem 4.3 *Let $\mathcal{S}$ be a given signature and $\mathscr{D}$ a corresponding safe operator specification. Then $\mathfrak{R} = \mathfrak{R}(\mathscr{D})$ is convergent.*

Proof. That $\mathfrak{R}$ is terminating is obvious because for every $\mathfrak{R}$-rule the right-hand side is strictly smaller than the left-hand side. A terminating relation is known to be convergent if it is locally confluent [20]. A term a exhibits local confluence if whenever

$$a \underset{\mathfrak{R}}{\rightarrow} b, a \underset{\mathfrak{R}}{\rightarrow} c \tag{4.12}$$

then

$$b \underset{\mathfrak{R}}{\rightarrow}^* d, c \underset{\mathfrak{R}}{\rightarrow}^* d \tag{4.13}$$

for some d. (See Figure 4.2.) A rewrite relation is then locally confluent iff all terms exhibit local confluence. We prove now that for $\mathfrak{R}$, every term exhibits local confluence.

Let t be any term. If $t \in \mathscr{F}_0$, then t is in its unique normal form and exhibits trivially local confluence. Therefore, assume $t = f(t_1, \dots, t_k)$. We denote by b_i ($i \in \{1, \dots, k\}$) the term $t[t']|_i$ when $t_i \underset{\mathfrak{R}}{\rightarrow} t'$ (thus when b_i exists, $t \underset{\mathfrak{R}}{\rightarrow} b_i$). We denote by R the set of those rules in $\mathfrak{R}$ whose left-hand sides match t as a whole. We denote by g_i ($i \in \{1, \dots, |R|\}$) the corresponding rewrited terms, i.e. the corresponding right-hand sides of the instances of the rules in R. Now, if $t \underset{\mathfrak{R}}{\rightarrow} t'$ then $t' \in \{b_1, \dots, b_k, g_1, \dots, g_{|R|}\} = Z$. We need now to show that for every pair $z, z' \in Z$, $(\exists u)[z, z' \underset{\mathfrak{R}}{\rightarrow}^* u]$. For every $z \in Z$, let $\mathbf{s}(z)$ denote the operator definition ($\in \mathscr{D}$) corresponding to the rule instance that was used to reduce t to z. We denote by $a \underset{\mathfrak{R}\,t}{\to} b$ the fact that $a \underset{\mathfrak{R}}{\to} b$ and that b is obtained as the right-hand side of a rule whose left-hand side was matched by a as a whole, i.e. that a reduces to b on the top level.

Now, if $z = b_i$ and $z' = b_j$, then trivially $z, z' \underset{\mathfrak{R}}{\to} f(t_1, \ldots, b_i, \ldots, b_j, \ldots, t_k)$. Therefore we may assume that $z = g_i$ for some i. Now there are five cases.

1. $\mathbf{s}(z) = P_{\text{inv}}(f, a)$: We have $t = f(a(z)) \underset{\mathfrak{R}\,t}{\to} z$. **(1)** Assume $z' = g_i$. Then $f(a(z)) \underset{\mathfrak{R}\,t}{\to} z'$ and necessarily either **(i)** $\mathbf{s}(z') = P_{\text{inv}}(f, a)$, thus $z = z'$, or **(ii)** $\mathbf{s}(z') = P_{\text{1-tuple}}(a, f)$, thus also $z = z'$. **(2)** Assume then $z' = b_i$. Then $f(a(z)) \underset{\mathfrak{R}}{\to} f(x) = z'$, where $a(z) \underset{\mathfrak{R}}{\to} x$. If **(i)** $a(z) \underset{\mathfrak{R}\,t}{\to} x$ then $z = v(x)$ and $a(v(x)) \underset{\mathfrak{R}\,t}{\to} x$ and $P_{\text{inv}}(a, v) \in \mathscr{D}$ from which by (CI) $v = f$ follows, thus $z, z' = f(x)$ ($P_{\text{1-tuple}}(v, a) \notin \mathscr{D}$ because of $P_{\text{inv}}(f, a)$ and (SNI)).

   On the other hand, **(ii)** if $f(z) \underset{\mathfrak{R}\,t}{\not\to} x$ then $f(x) \underset{\mathfrak{R}}{\to} f(u) = x$ and $z \underset{\mathfrak{R}}{\to} u$. We get $z' = f(a(u))$ and finally $z, z' \underset{\mathfrak{R}}{\to} u$.

2. $\mathbf{s}(z) = P_{n\text{-tuple}}(c, f_1, \ldots, f_n)$ and $f = f_\ell$: Necessarily $t = f(c(t_1, \ldots, t_k))$ and $k = n$. We have $t \underset{\mathfrak{R}\,t}{\to} t_\ell = z$. **(1)** Assume $z' = g_i$. Then either **(i)** $\mathbf{s}(z') = P_{n\text{-tuple}}(c, f'_1, \ldots, f'_n)$, $f = f'_j$, thus by (MS) $j = \ell$ and $z' = t_\ell = z$, **(ii)** or $\mathbf{s}(z') = P_{\text{inv}}(f, c)$ and $n = 1$, thus $\ell = 1$ and $z' = t_1 = z$.**(2)** Assume then $z' = b_i$. Then by (CDNR) $z' = f(c(t_1, \ldots, t', \ldots, t_k))$. If t' is not the ℓth subterm of $c(\ldots)$, then $z' \underset{\mathfrak{R}}{\to} t_\ell = z$. If t' is the ℓth subterm, then $t_\ell \underset{\mathfrak{R}}{\to} t'$. Therefore $z', z \underset{\mathfrak{R}}{\to} t'$.

3. $\mathbf{s}(z) = P_{\text{symenc}}(e, f)$: We have $t = f(x, e(x, z)) \underset{\mathfrak{R}\,t}{\to} z$. **(1)** Assume $z' = g_i$. Then by (SAASD) $\mathbf{s}(z') = P_{\text{symenc}}(e, f)$ and $z' = z$. **(2)** Assume then $z' = b_i$. Then there are three cases. **(i)** $z' = f(x', e(x, z))$ or $z' = f(x, e(x', z))$. Then $x \underset{\mathfrak{R}}{\to} x'$, therefore $z' \underset{\mathfrak{R}}{\to} f(x', e(x', z)) \underset{\mathfrak{R}}{\to} z$. **(ii)** $z' = f(x, e(x, z''))$. Then $z, z' \underset{\mathfrak{R}}{\to} z''$. **(iii)** $z' = f(x, u)$ where $e(x, z) \underset{\mathfrak{R}\,t}{\to} u$. Then by (SAASD) $\mathbf{s}(e(x, z)) = P_{\text{symenc}}(a, e)$ and $z = a(x, u)$ By (CSS), $a = f$. Thus $z = f(x, u) = z'$

4. $\mathbf{s}(z) = P_{\text{pubenc}}(e, f, p, s)$: We have $t = f(s(k), e(p(k), z)) \underset{\mathfrak{R}}{\to} z$. **(1)** Assume $z' = g_i$. Then by (SAASD) $\mathbf{s}(z') = P_{\text{pubenc}}(e, f, p, s)$; thus $z' = z$. **(2)** Assume then $z' = b_i$. There are four cases. **(i)** $z' = f(s(k'), e(p(k'), z))$ or $z' = f(s(k), e(p(k'), z))$. Then $k \underset{\mathfrak{R}}{\to} k'$, therefore $z' \underset{\mathfrak{R}}{\to} f(s(k'), e(p(k'), z)) \underset{\mathfrak{R}}{\to} z$. **(ii)** $s(k) \underset{\mathfrak{R}\,t}{\to} u$ and $p(k) \underset{\mathfrak{R}\,t}{\to} u$ are

impossible by (SKDF). **(iii)** $z' = f(s(k), e(p(k), z''))$. Then $z, z' \underset{\mathfrak{R}}{\to} z''$. We are left with **(iv)** $z' = f(s(k), u)$ where $e(p(k), z) \underset{\mathfrak{R}\,t}{\to} u$. Then by (SAASD) $\mathbf{s}(e(p(k), z)) = P_{\text{pubenc}}(a, e, b, p)$ and $z = a(b(k), u)$ and $z' = f(s(k), u)$. By (CAS) $f = a$ and $s = b$, thus $z = z'$.

5. $\mathbf{s}(z) = P_{\text{signs}}(c, f, p, s)$: We have $t = f(m, p(k), c(s(k), m))$ and $z = 1$. **(1)** Assume $z' = g_i$. Then $\mathbf{s}(z') = P_{\text{signs}}(c, f, p, s)$; thus $z' = z$. **(2)** Assume then $z' = b_i$. There are four cases. **(i)** $z' = f(m', p(k), c(s(k), m))$ or $z' = f(m, p(k), c(s(k), m'))$. Then $z' \underset{\mathfrak{R}}{\to} f(m', p(k), c(s(k), m')) \underset{\mathfrak{R}}{\to} 1$. **(ii)** $z' = f(m, p(k'), c(s(k), m))$ or $z' = f(m, p(k), c(s(k'), m))$. Then $z' \underset{\mathfrak{R}}{\to} f(m, p(k'), c(s(k'), m)) \underset{\mathfrak{R}}{\to} 1$. **(iii)** $s(k) \underset{\mathfrak{R}\,t}{\to} u$ and $p(k) \underset{\mathfrak{R}\,t}{\to} u$ are impossible by (SKDF). We are left with **(iv)** $z' = f(m, p(k'), u)$ where $c(s(k'), m) \underset{\mathfrak{R}\,t}{\to} u$. But this case is impossible because of (SND).

Together, these cases cover all the possibilities. □

4.1.3 Model Drawn Together

We are now able to summarize the model we use. $\mathcal{S}$, a signature, must be chosen first. The elements of $\mathscr{F}_0$ denote *atomic messages*, from which terms are built. Because they are very important they deserve a notation of their own. We thereby denote later $\mathcal{A} = \mathscr{F}_0$ in order to emphasize the special nature of this set. After $\mathcal{S}$ has been chosen, a safe operator specification $\mathscr{D}$ must be given. Together they define the algebra $\mathscr{M}(\mathcal{S}, \mathfrak{R}(\mathscr{D}))$. We will denote this algebra by $\mathscr{M}$ when $\mathcal{S}$ and $\mathscr{D}$ can be assumed fixed. $\mathscr{M}$ is now an algebra homomorphic to a particular cryptoalgebra and works as the model for cryptography.

In this chapter, we will work under $\mathscr{M}$ unless otherwise stated. The semantical braces will be dropped when no confusion should arise. This is done to simplify notation. Also, we will identify the elements of $\mathcal{S}$ with those of $\mathscr{F}_{\mathscr{M}}$, i.e. identify the operator names with the corresponding functions.

Definition 4.6 Let $M \subseteq \mathscr{M}$ and $m \in \mathscr{M}$. We use $M \vdash m$ to denote $m \in M^{*[\mathscr{M}]}$ in order to emphasize the fact that m belongs to M^* iff it can be derived from the set M. □

4.2 Protocols and Semantics

In this section we define the language used to describe the workings of individual participants. Also, we define the semantics of the programs denoted by the language. These semantics incorporate also the threat model.

We give the semantics in a standard fashion. There are n participants that communicate via the environment, which is controlled by the attacker. Every participant has its own local state. The environment has also a state. Together the states of the environment and the participants form the global state of the whole system. This global state evolves partly nondeterministically and partly deterministically. The deterministic part comes from the participants that follow their programs. The nondeterministic part is caused by the environment. This is because the attacker cannot be assumed to work in any specific way a priori. Therefore the attacker is modeled by a nondeterministic process.

The programs the participants execute are loopless programs. They are finite sequences that have conditional branches.

The local state of a participant consists of the program to execute and a set of assignments to state variables. Every state variable can be assigned to one element of $\mathscr{M}$ or have an undefined value. The number of state variables available is in principle infinite. However, at the start of the program, all variables have an undefined value. Because the programs are finite only a finite subset of the variables can ever be assigned.

The actions the participants are able to execute (and thus the actions that the programs can describe) are the following: **(1)** receiving a message from the environment; **(2)** sending a message to the environment; **(3)** generating locally a *nonce*, a large random number; **(4)** comparing two state variables and branching depending on the result; **(5)** binding a state variable to a message composed of the values of some other state variables and some constant messages; and **(6)** halting.

The next subsections give formal definitions.

4.2.1 Programs

Definition 4.7 Let $\mathcal{V}$ be a fixed set of symbols, namely the *state variables*. We define the set $\mathcal{P}_*$ of *programs* as the smallest set containing the following, for any $p, p' \in \mathcal{P}_*$, $v, v' \in \mathcal{V}$, $n \in \mathbb{N}$ and $t \in \mathbf{T}_{\mathcal{S}}(\mathcal{V})$:

1. $\mathbf{halt}_n$ ('halt'),

2. $\Lsh v.p$ ('send v and then p'),
3. $\Rsh v.p$ ('receive v and then p'),
4. $\uparrow v.p$ ('generate v and then p'),
5. $(v = v')?p : p'$ ('p if $v = v'$, otherwise p''), and
6. $v \leftarrow t.p$, ('let v be t in p'). □

The number n in $\mathbf{halt}_n$ is for purely technical reasons. It is used to distinguish different halting states from each others and to ease later development in general.

Definition 4.8 Let $p \in \mathcal{P}_*$. The set of *immediate subprograms* of p, $\mathrm{sub}_0(p)$, is defined as

$$\mathrm{sub}_0(\mathbf{halt}_n) = \emptyset \tag{4.15}$$
$$\mathrm{sub}_0(\Lsh v.p) = \{p\} \tag{4.16}$$
$$\mathrm{sub}_0(\Rsh v.p) = \{p\} \tag{4.17}$$
$$\mathrm{sub}_0(\uparrow v.p) = \{p\} \tag{4.18}$$
$$\mathrm{sub}_0((v = v')?p : p') = \{p, p'\} \tag{4.19}$$
$$\mathrm{sub}_0(v \leftarrow t.p) = \{p\}. \tag{4.20}$$

The set of *subprograms* of p is denoted by $\mathrm{sub}(p)$ and is defined by

$$\mathrm{sub}(p) = \{p\} \cup \left(\bigcup_{p' \in \mathrm{sub}_0(p)} \mathrm{sub}(p') \right).\Box \tag{4.21}$$

Definition 4.9 Let $p \in \mathcal{P}_*$. We say that v is *assigned in* p iff p is of one of the following forms: $\Rsh v.p'$, $\uparrow v.p'$ or $v \leftarrow t.p'$. □

Definition 4.10 Let $p \in \mathcal{P}_*$. p is said to be a *valid program* if the following requirements are fulfilled:

1. If v is assigned in p, then v is not assigned in any $p' \in \mathrm{sub}(p) \setminus p$.
2. For all $n \in \mathbb{N}$, $\mathbf{halt}_n$ appears in p at most once.
3. If $v \leftarrow t.p' \in \mathrm{sub}(p)$ and v' is a variable appearing in t, then there exists p'' such that $p'' \in \mathrm{sub}(p)$ and $v \leftarrow t.p' \in \mathrm{sub}(p'')$ and that v' is assigned in p''.

The set of valid programs is denoted by $\mathcal{P}$. □

The valid programs can be seen to be programs in a 'low-level' programming language that describe the workings of individual protocol participants. The programs can be read as follows: $\mathbf{halt}_n$ specifies that the participant should just stop. The natural number n is used to distinguish unambiguously between different halting states and thus also between program branches in general. $\Lsh\, v.p$ specifies that the participant should send to the environment the contents of the variable v and then continue with the program p. Similarly, $\Rsh\, v.p$ specifies that the participant should receive a message from the environment and store it to the variable v. $\uparrow v.p$ specifies that the participant should generate a nonce and stored it to v. $v \leftarrow t.p$ specifies that the participant should store the value of the term t to the variable v. Here the term t may contain instances of state variables, i.e their names; they are replaced with their values at the moment of the assignment. $(v = v')?p : p'$ is a conditional construct. If the contents of v and v' are equivalent the participant should continue with the program p, otherwise p'.

The extra requirements imposed on a valid program translate to the following ideas (numbering corresponds to that of Definition 4.10): **(1)** Every variable will be assigned at most once during any execution. **(2)** All halting states are distinct. **(3)** Terms t appearing in programs of the form $v \leftarrow t.p$ have fully specified values, i.e. contain only names of those state variables that have already been assigned to.

Example 4.4 Consider the simplified Needham-Schröder public key protocol using RSA (see Chapter 10 and Appendix B):

$$\begin{array}{lllll}
(1) & A & \longrightarrow & B & : \mathsf{RSAenc}(\mathsf{RSApubk}(k_B), \mathsf{cons}(A, n_A)) \\
(2) & B & \longrightarrow & A & : \mathsf{RSAenc}(\mathsf{RSApubk}(k_A), \mathsf{cons}(n_A, n_B)) \\
(3) & A & \longrightarrow & B & : \mathsf{RSAenc}(\mathsf{RSApubk}(k_B), n_B)
\end{array}$$

Here the corresponding operating specification could be

$$\mathscr{D} = \{P_{\text{pubenc}}(\mathsf{RSAenc}, \mathsf{RSAenc}, \mathsf{RSApubk}, \mathsf{RSAprivk}), P_{\text{2-tuple}}(\mathsf{cons}, \mathsf{car}, \mathsf{cdr})\}. \tag{4.22}$$

Thus, cons creates 2-tuples whose elements can be accessed by using car and cdr. (The names are those traditionally used in the LISP community for the pair constructor and the pair selectors.) RSAenc obviously denotes RSA encryption. (Although the protocol could of course use some other public-key cryptosystem as well.)

Let **mynonce**, **hernonce**, **mynonceback**, **msg1**, **msg2** and **msg3** be state variables. Then the program p for the participant A in the protocol above can be given as

$$\begin{aligned} p = {} & \uparrow \mathbf{mynonce}. \\ & \mathbf{msg1} \leftarrow \mathsf{RSAenc}(\mathsf{RSApubk}(k_B), \mathsf{cons}(A, \mathbf{mynonce})). \\ & \Lsh \mathbf{msg1}. \\ & \Rsh \mathbf{msg2}. \\ & \mathbf{mynonceback} \leftarrow \mathsf{car}(\mathsf{RSAdec}(\mathsf{RSAprivk}(k_A), \mathbf{msg2})). \\ & (\mathbf{mynonce} = \mathbf{mynonceback})?p' : \mathbf{halt}_1 \end{aligned}$$

where

$$\begin{aligned} p' = {} & \mathbf{hernonce} \leftarrow \mathsf{cdr}(\mathsf{RSAdec}(\mathsf{RSAprivk}(k_A), \mathbf{msg2})). \\ & \mathbf{msg3} \leftarrow \mathsf{RSAenc}(\mathsf{RSApubk}(k_b), \mathbf{hernonce}). \\ & \Lsh \mathbf{msg3}.\mathbf{halt}_2. \end{aligned}$$

This program reads as: 'Generate a new value for **mynonce**. Then construct a message that has the name A and the value of **mynonce** *cons*ed together and then encrypted with the public key of B. Send this message. Receive another message. Extract the first field of the received message after it has been decrypted using the private key of A. Store this field to the variable **mynonceback**. Compare the values of **mynonce** and **mynonceback**. If they differ, halt immediately. Otherwise extract the second field of the decrypted message to **hernonce**. Encrypt **hernonce** using B's public key to **msg3**. Send **msg3** and halt.' □

4.2.2 Semantics of Protocols

The intuitive semantics of programs was outlined in the previous subsection. Here the formal semantics of a system composed of many participants and an environment is defined.

Definition 4.11 A *local state* is a triple $\langle p, \Upsilon, c\rangle$ where $p \in \mathcal{P}$, $\Upsilon \in \mathbf{subst}(\mathcal{V}, \mathbf{T}_{\mathcal{S}}(\emptyset))$ and $c \in \mathbb{N}$. The set of all local states is denoted by $\mathcal{L}$. □

In the above definition p is the program the participant is executing, Υ represents the contents of the state variables and c is a counter that is used to create unique atomic messages that denote the nonces generated by the participant.

Definition 4.12 Let $\mathcal{N}$ be a fixed finite set of *participant names*. A global state is a pair $\langle \mathscr{L}, M\rangle$ where $\mathscr{L}$ is a total function from $\mathcal{N}$ to $\mathcal{L}$ and M is a finite subset of $\mathscr{M}$. The set of global states is denoted by $\mathcal{G}$. The semantics of protocols are implemented in the terms of the labeled transition relation $\underset{\ell}{\rightarrow}$ between global states, defined as follows: $\langle \mathscr{L}, M\rangle \underset{\ell}{\rightarrow} \langle \mathscr{L}', M'\rangle$ iff $\mathscr{L}' = \mathscr{L}[n \mapsto \langle p', \Upsilon', c'\rangle]$ for some $n \in \mathcal{N}$, $\mathscr{L}(n) = \langle p, \Upsilon, c\rangle$, $\ell = n.p$ and one of the following holds:

1. p is $\Lsh\, v.p'$, $\Upsilon' = \Upsilon$, $M' = M \cup \Upsilon(v)$ and $c' = c$.
2. p is $\Rsh\, v.p'$, $\Upsilon' = \Upsilon[v \mapsto m]$ for some $m : M \vdash m$, $M' = M$ and $c' = c$.
3. p is $\uparrow v.p'$, $\Upsilon' = \Upsilon[v \mapsto n.c]$, $M' = M$ and $c' = c + 1$.
4. p is $v \leftarrow t.p'$, $\Upsilon' = \Upsilon[v \mapsto \Upsilon(t)]$, $M' = M$ and $c' = c$.
5. p is $(v = v')?p' : x$, $\Upsilon(v) = \Upsilon(v')$, $\Upsilon' = \Upsilon$, $M' = M$ and $c' = c$.
6. p is $(v = v')?x : p'$, $\Upsilon(v) \neq \Upsilon(v')$, $\Upsilon' = \Upsilon$, $M' = M$ and $c' = c$.

(Recall that we work under $\mathscr{M}$. This is, the equalities and inequalities in the items 5 and 6 correspond to the congruence of (4.4) over all the ground terms.)

The combined relation $\bigcup_\ell \underset{\ell}{\rightarrow}$ is denoted by $\rightarrow$. If there is no g' such that $g \rightarrow g'$, g is said to be a *terminal state*. The set of all terminal global states is denoted by $\mathcal{G}_T$.

This definition assumes that all nonces belong to $\mathcal{A}$ (the set of atomic messages), i.e. if $n \in \mathcal{N}$ and $c \in \mathbb{N}$ then $n.c \in \mathcal{A}$. □

4.3 Analysis

Definition 4.13 Let $T = 2^{\mathcal{G}_T}$ be the set of predicates defined over the terminal global states. We define the predicate **safety** $\subseteq \mathcal{G} \times T$ as

$$\mathbf{safety}(g, \tau) \quad \text{iff} \quad (\forall g' \in \mathcal{G}_T)[g \rightarrow^* g' \Rightarrow \tau(g')].\square \tag{4.23}$$

Intuitively, $\mathbf{safety}(g, \tau)$ is true exactly if there is no terminal state that can be reached from g and that does not fulfill τ. The name of the predicate is motivated by the fact that the terminal predicates τ will be used in this work to describe security assumptions. Therefore, $\mathbf{safety}(g, \tau)$ will be true exactly if no states violating the security assumptions can be reached from g.

function SEARCH(g)
  if g is a terminal state **then**
    if $\neg\tau(g)$ holds **then**
      Report '**safety** = FALSE'. Stop.
    end if
  else
    for all $g' : g \to g'$ **do**
      SEARCH(g')
    end for
    If g is the initial state, report '**safety** = TRUE' and stop.
  end if

Figure 4.3 A pseudo-algorithm for enumerating a state space.

In the context of this work, the analysis of cryptographic protocols corresponds to the problem of deciding the truth value of **safety**(g, τ) for a given g and a given τ. Given that τ is a suitable, computable predicate, state space enumeration can be used to find this value.

4.3.1 State Space Enumeration

The idea of state space enumeration is to explicitly enumerate the set of reachable states from a given initial state g, i.e. the set

$$\{g' \mid g \to^* g'\}. \tag{4.24}$$

Such an enumeration will eventually also enumerate all the terminal states reachable from g. Then the value of **safety** can be computed by checking if the set of the enumerated terminal states contains a state for which τ does not hold. A naïve recursive algorithm for enumerating all the terminal states reachable from g and calculating the value of **safety** is presented in Figure 4.3. For our problem the state space is loopless. Therefore we do not need to check for already visited states in the algorithm (although that could be more efficient).

However, the problem with this algorithm and the approach in general is that the set $\{g' \mid g \xrightarrow[n.\Rsh v.p]{} g'\}$ is infinite if $M \neq \emptyset$ in g (M is the set of messages contained in the environment). This is because the semantics of protocols are defined so that any term m derivable from a given set can be assigned to the state variable v when a message is received, *and there are infinitely many choices.* Therefore the algorithm never terminates!

function SSEARCH($\hat{g}$)
  if $\rho(\hat{g})$ is a set of terminal states **then**
    if $\neg\tau_s(\hat{g})$ holds **then**
      Report '**safety** = FALSE'. Stop.
    end if
  else
    for all $\hat{g}' : \hat{g} \hookrightarrow \hat{g}'$ **do**
      SSEARCH($\hat{g}'$)
    end for
    If $\hat{g}$ is the initial symbolic state, report '**safety** = TRUE' and stop.
  end if

Figure 4.4 A pseudo-algorithm for symbolically enumerating a state space.

4.3.2 Symbolic State Space Enumeration

However, due to the semantics, all traces

$$g \to \cdots \to g'$$

are of finite length. Therefore a symbolic method can succeed as follows. Let R be a set and ρ a function $\rho : R \mapsto 2^{\mathcal{G}}$. Here R is a set of *state set representations* and ρ a *representation function*. Let τ be a given predicate over the terminal global states. Let $\hat{\tau}$ be the predicate over R defined as

$$\hat{\tau}(\hat{g}) \quad \text{iff} \quad (\forall g \in \rho(\hat{g}))[g \in \mathcal{G}_T \Rightarrow \tau(g)]. \tag{4.25}$$

Let $\hookrightarrow$ be a relation over $R \times R$ such that first

$$\hat{g} \hookrightarrow \hat{g}' \Rightarrow (\forall g' \in \rho(\hat{g}'))[(\exists g \in \rho(\hat{g}))[g \to g']]. \tag{4.26}$$

Denoting the set $\{\hat{g}' \mid \hat{g} \hookrightarrow \hat{g}'\}$ by $s(\hat{g})$, let further $s(\hat{g})$ be always finite and let it hold that

$$g \in \rho(\hat{g}) \wedge g \to g' \Rightarrow (\exists \hat{g}' \in s(\hat{g}))[g' \in \rho(\hat{g}')] \tag{4.27}$$

Then the algorithm in Figure 4.4 can be used to the end that the previous algorithm failed to achieve; this algorithm performs a symbolic enumeration of the infinite state space in a finite number of steps in the sense that the value of **safety**(g, τ) will be correctly calculated by SSEARCH($\hat{g}$) whenever $\rho(\hat{g}) = \{g\}$.

The next chapter focuses on implementing a symbolic representation for our problem that fulfills the propertied given in (4.25)–(4.27).

4.3.3 Terminal Predicates

The generic statement of the analysis problem (see page 35) includes a predicate τ over terminal states and a corresponding predicate $\hat{\tau}$ over symbolic terminal states that are bound together by (4.25). However, until now all (computable) predicates over $\mathcal{G}_T$ have been considered acceptable. While a generic computable predicate could work with a simple state space enumeration this is not necessarily the case with the symbolic version. The problem is that even if τ were be computable, it might be that $\hat{\tau}$ were not.

The question about the computability of $\hat{\tau}$ of course depends also on the choice of the symbolic representation in general. While our representation is the subject of the next chapter, we now immediately define a class of terminal predicates that we will confine ourselves to.

Definition 4.14 Let $\mathcal{N}$ denote the set of participant names, $\mathscr{X}_P$ a set of *participant variables* such that $\mathscr{X}_P \cap \mathcal{N} = \emptyset$, and let $\mathcal{V}$ denote the set of state variables. The set $(\mathcal{N} \cup \mathscr{X}_P) \times \mathcal{V}$ is the set of *qualified state variables.* This set is denoted by $\mathcal{V}_{\mathcal{N}\cup\mathscr{X}_P}$. An element $\langle n, v\rangle$ of $\mathcal{V}_{\mathcal{N}\cup\mathscr{X}_P}$ will be written $n : v$. The subset $\mathcal{N} \times \mathcal{V}$ of $\mathcal{V}_{\mathcal{N}\cup\mathscr{X}_P}$ is the set of *constant-qualified state variables* and is denoted by $\mathcal{V}_{\mathcal{N}}$. The largest subset of $\mathcal{V}_{\mathcal{N}\cup\mathscr{X}_P}$ that does not contain any occurrences of $\mathscr{X}_P \setminus P$, where $P \subseteq \mathscr{X}_P$, is denoted by $\mathcal{V}_{\mathcal{N}\cup P}$. □

Definition 4.15 We define the function $\tilde{\Upsilon} : \mathcal{G} \times \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}}) \mapsto \mathbf{T}_{\mathcal{S}}(\emptyset)$ that maps terms containing zero or more constant-qualified state variables to ground terms as follows. Let $g = \langle \mathscr{L}, M\rangle$. Then

$$\tilde{\Upsilon}(g,t) = \begin{cases} t, & \text{if } t \in \mathbf{T}_{\mathcal{S}}(\emptyset) \\ \mathscr{L}_2(n)(v), & \text{if } t = n : v \\ f(\tilde{\Upsilon}(g,t_1), \dots, \tilde{\Upsilon}(g,t_k)) & \text{otherwise when } t = f(t_1,\dots,t_k). \end{cases} \tag{4.28}$$

(Recall that $\mathscr{L}_2(n)(v)$ is the value of v in the principal n's valuation.) □

Definition 4.16 We define the language Φ_τ as the language generated by the following infinite grammar starting from the initial symbol $\emptyset$ (all nonterminals are subsets of $\mathscr{X}_P$). The actual symbols of the language are underlined to distinguish them from the symbols used to represent the grammar, e.g. the set union on the last two lines:

$$S \to \underline{\top} \tag{4.29}$$
$$S \to S\underline{\wedge}S \tag{4.30}$$
$$S \to S\underline{\vee}S \tag{4.31}$$
$$S \to \underline{\neg}S \tag{4.32}$$
$$S \to \underline{t = t'} \quad \text{where } t, t' \in \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}\cup S}) \tag{4.33}$$
$$S \to \underline{\mathbf{S}(t)} \quad \text{where } t \in \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}\cup S}) \tag{4.34}$$
$$S \to \underline{\forall r}(S \cup \{r\}) \quad \text{where } r \in \mathscr{X}_P \tag{4.35}$$
$$S \to \underline{\exists r}(S \cup \{r\}) \quad \text{where } r \in \mathscr{X}_P \tag{4.36}$$

The language Φ_τ is exactly the syntactical set of the terminal predicates we allow. Thus, a terminal predicate is an element of Φ_τ. The next definition gives semantics for these terminal predicates. □

Definition 4.17 Let $g \in \mathcal{G}$ be a global state. We denote by $g \models \tau$ the fact that τ is true for the given global state. This relation is defined as follows:

1. $g \models \underline{t = t'}$ iff $\tilde{\Upsilon}(g, t) = \tilde{\Upsilon}(g, t')$
2. $\langle \mathscr{L}, M \rangle \models \underline{\mathbf{S}(m)}$ iff $M \not\vdash m$
3. $g \models \underline{\top}$ always
4. $g \models \underline{\neg\tau}$ iff not $g \models \underline{\tau}$
5. $g \models \underline{\tau \wedge \tau'}$ iff $g \models \underline{\tau}$ and $g \models \underline{\tau'}$
6. $g \models \underline{\tau \vee \tau'}$ iff $g \models \underline{\tau}$ or $g \models \underline{\tau'}$
7. $g \models \underline{\forall r \tau}$ iff $g \models \underline{\tau'}$ for all τ' that have been obtained by replacing the free occurrences of r in the qualified state variables appearing τ with an $n \in \mathcal{N}$.
8. $g \models \underline{\exists r \tau}$ iff $g \models \underline{\tau'}$ for some τ' that has been obtained by replacing the free occurrences of r in the qualified state variables appearing in τ with an $n \in \mathcal{N}$. □

We give an implementation of $\hat{\tau}$ corresponding to this definition in the next chapter along with the symbolic state space enumeration method.

Chapter 5

Symbolic Representation

A symbolic state space representation is defined. Algorithms for manipulating the symbolic states are developed.

As mentioned in the previous chapter, a symbolic method is necessary for the enumeration of an infinite state space. In this chapter, we first introduce symbolic global states, defined with the help of *constraint sets*. We develop a procedure for solving constraint sets and reducing them to 'smaller' sets. We then give a procedure for symbolic state space enumeration.

5.1 Intuitive Overview

The machinery contained in this chapter may seem complicated at first. Therefore we dedicate one section for giving an intuitive overview of the contents.

5.1.1 Symbolic States

Symbolic states are objects that denote zero or more nonsymbolic global states. This relation is mediated by a representation function that maps a symbolic state to the set of nonsymbolic states it denotes. The resulting set can be either finite or countably infinite. This relation between the symbolic states and the nonsymbolic ones is illustrated in Figure 5.1. Observe that one of the symbolic states actually represents no nonsymbolic states. Thus it is mapped by the representation function to the empty set.

In our work symbolic states can be seen to consist of two parts.

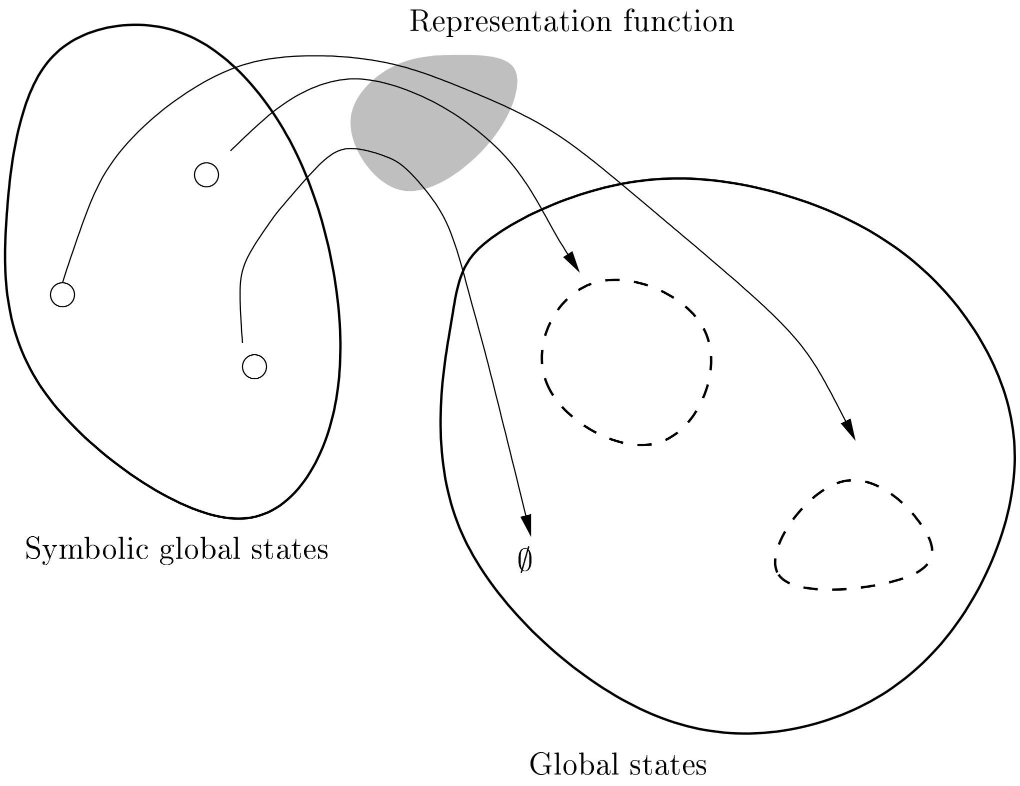

Figure 5.1 Symbolic and nonsymbolic global states.

The first part is a state template that looks like a nonsymbolic global state. However, a state template may also contain *variables* that can denote a multitude of different terms. Such variables can appear in two places: in the values of the principals' state variables and in the set of messages known by the environment (recall Definition 4.12).

The second part is a constraint set. The constraint set is a mathematical object that limits the choices of terms that the variables in the template can actually denote. This works as follows. For any substitution that maps all the variables that occur in the template and in the constraint set (the constraint set contains the same variables) to ground terms, the constraint set becomes either *contradictory* or not when the substitution is applied to it. A substitution that causes the set to be not contradictory is said to be a *realizing substitution* and a *solution* of the constraint set. Applying a realizing substitution to the corresponding template, i.e. replacing the variables in the template with the values from the substitution, gives a nonsymbolic global state that is one of the states represented by the symbolic state.

The symbolic state represents exactly all those global states that are obtained from the template by applying to it a realizing substitution, i.e. a substitution that makes the constraint set noncontradictory. This internal structure of a symbolic state is illustrated in Figure 5.2: every particular solution to the constraint set gives one global state that belongs to the set of states denoted by the symbolic state.

The actual reason why a symbolic method works well for our problem is that typically there is an infinite number of global states that are 'similar enough' to be bundled together. For such classes, we are not interested in the individual states represented but only in the class as a whole. *A symbolic state denotes a class of somehow similar states.* But even though we are not interested in the individual members of the class, we would like be able to know whether or not the class is *empty*. In other words, we need to be able to detect such symbolic states that actually do not represent any global state at all, i.e. those states whose constraint sets do not have any solutions. This is because such symbolic states should be dropped from the search to avoid unnecessary work.

For this purpose we will introduce a transformation on symbolic states that maps one symbolic state to one or more symbolic states in a specific way. Namely, the resulting symbolic states together will denote exactly the same states as the original symbolic state. A transformation like this is illustrated in Figure 5.3.

At first glance it might not be clear why it is useful to split a symbolic state into many ones. However, the transformation we use will have the nice property that the symbolic states become in a certain way simpler when they are

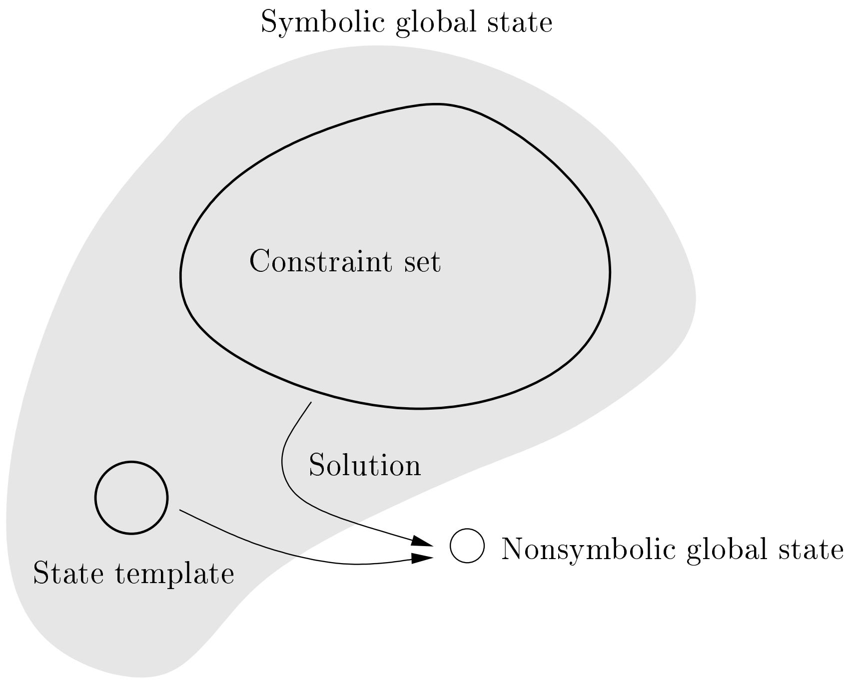

Figure 5.2 The structure of a symbolic state.

split. As the transformation is iterated over and over again, the symbolic states that eventually remain are all in a simple, canonical form. The useful property of the canonical form is that *every symbolic state in the canonical form represents a non-empty set of nonsymbolic states.* It can now be deduced that the transformation also detects those states that have contradictory constraint sets, i.e. those that do not represent any nonsymbolic states. Such states can be immediately discarded.

Thus, when iterated enough times the transformation maps any symbolic state to a set of symbolic states in the canonical form. It is possible that this set is empty. Then, and only then, the original state as a whole did not denote any nonsymbolic state.

The symbolic states that are generated during the analysis process have also the following property: any solution to a state's constraint set, a substitution, is such that the resulting nonsymbolic global state contains only normal-form terms. (Applying a substitution does not involve the rewrite system, and therefore in principle terms not in their normal forms could be generated.) This property does not however hold trivially. Instead, the whole process is designed so that the property holds inductively. This should be kept in mind.

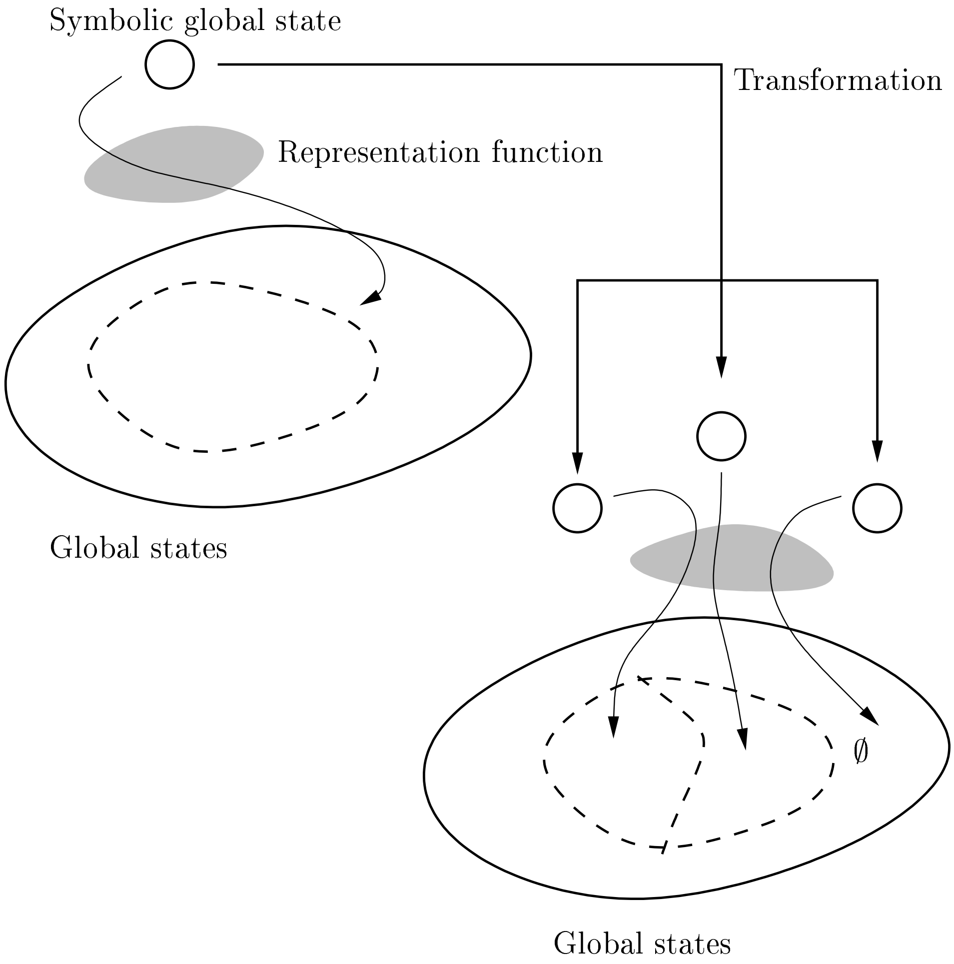

Figure 5.3 A one-to-many transform of a symbolic state.

5.1.2 Bases

The semantics of the receive action (recall Definition 4.12) involve the set of messages derivable from a given set $M \subseteq \mathscr{M}$, i.e. the algebraic closure of M under $\mathscr{M}$. Deciding whether or not $M \overset{?}{\vdash} m$ is relatively straightforward for ground terms but the issue becomes more complicated when both m and M can contain variables. When they do we cannot expect a general 'yes' or 'no' answer. Instead, we want to know the set of grounding substitutions σ so that $\sigma(M) \vdash \sigma(m)$.

In order to cope with the problem of deciding this kind of derivability we introduce the concept of a *basis*. A basis of a ground term set M is a ground term set B such that every message that belongs to the algebraic closure of M under $\mathscr{M}$ belongs also to the algebraic closure of B under $\mathbf{F}(\mathcal{S})$. The reverse also hold for *normal form* terms (the algebraic closure of B under $\mathbf{F}(\mathcal{S})$ can contain also terms not in their normal forms, and thus terms not in $\mathscr{M}$). In essence, if m can be composed from M by perhaps applying the rewrite system, m can be composed of the terms in B without involving the rewrite system.

Then the question of $M \overset{?}{\vdash} m$ can be reduced to first computing a basis of M and then (easily) checking whether or not m can be composed of the terms in B, without the rewrite system at all.

A corresponding notion for terms with variables will be that of a quasi-basis. If M is a set of terms with variables and Q is a quasi-basis of it, then the following holds: for every substitution σ that maps the variables in M to ground terms and makes M a set of terms in their normal forms, $\sigma(Q)$ is either a basis of $\sigma(M)$ or can be extended to one by adding more terms. We will use largest quasi-bases possible, in a sense, namely those that are genuine bases at least for some substitutions.

We will show later that every ground term set has a unique smallest basis. Similarly, every non-ground term set has a unique most generic and smallest quasi-basis.

5.1.3 Procedural Structure

The main algorithm of this chapter is the algorithm that implements the transformation on symbolic states that was introduced previously. The algorithm is split into the five procedures IN-CLOSURE?, QUASI-BASIS, EXPAND, REDUCE-STEP and REDUCE. The call hierarchy of the procedures is illustrated in Figure 5.4.

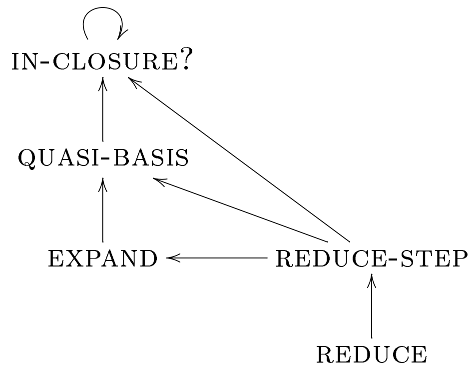

Figure 5.4 The procedures used to implement the reducing transformation.

REDUCE is the top-level procedure. It gets as its input a symbolic global state and returns a set of symbolic global states denoting the same set of nonsymbolic states as the original input denoted. This set may be empty if the input does not denote any real states at all.

The transformation performed by REDUCE is done by iterating a simple transformation implemented by REDUCE-STEP. REDUCE-STEP takes as its input a symbolic state and returns a set of symbolic states that, again, denote the same set of nonsymbolic states as the input denoted. REDUCE iterates REDUCE-STEP until a fixpoint has been found. Namely, REDUCE-STEP returns the singleton set containing the unmodified input exactly when the input is initially in the canonical form. All symbolic states eventually either transform to the canonical form or are detected contradictory. Thus, when REDUCE returns, it returns a set of canonical symbolic states.

QUASI-BASIS is a function computing the most generic and smallest quasi-basis of a given set of terms. IN-CLOSURE? decides, for a given M and m, whether or not $\sigma(m)$ belongs to the free closure (closure under $\mathbf{F}(\mathcal{S})$) of $\sigma(M)$ for all substitutions σ. Especially for ground terms and sets, IN-CLOSURE?(M, m) is true when $m \in M^{*[\mathbf{F}(\mathcal{S})]}$.

The last procedure, EXPAND, is called from REDUCE-STEP when there is a set M of terms such that for the solutions σ of the constraint set, $\sigma($QUASI-BASIS$(M))$ can both be and not be a basis of $\sigma(M)$. EXPAND is used to split the symbolic state containing M and the constraints to a set of states so that this uncertainty no longer exists in any of the resulting states.

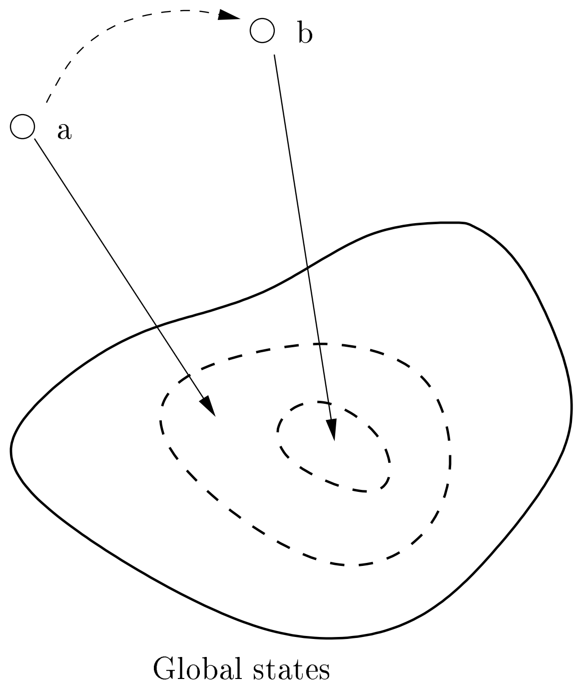

Figure 5.5 Symbolic state subsumption.

5.1.4 Having Only Normal-form Terms

As mentioned before, applying any solution of the constraint set of a symbolic state to the state template part results in a state that contains only normal-form terms, i.e. those that really belong to $\mathscr{M}$. We need to take care of this explicitly at some points of the symbolic state space enumeration process. This is done by the procedure ENFORCE. Basically, given a term and a constraint set, the procedure returns a collection of term and constraint set pairs such that for every such pair, every solution of the corresponding constraint set gives a normal-form term when applied to the term in the pair.

The procedure ENFORCE is not actually used in this chapter, but only in the next that describes the actual enumeration procedure.

5.1.5 Symbolic State Subsumption

It is possible that a symbolic state denotes a subset of those states denoted by some other symbolic state. This situation is illustrated in Figure 5.5. In this figure, the symbolic state a is said to *subsume* b.

In this chapter we also develop a simple algorithm that will in some cases detect symbolic state subsumption relations. The algorithm is not complete in the sense that some subsumption relations are actually not detected. This is not fatal, though, because the relations are used in the next chapter to speed up the state space enumeration. The enumeration procedure would remain valid even if no relations were detected.

5.1.6 Symbolic Terminal Predicates

The last remaining issue in this chapter is that of the symbolic terminal predicates. As mentioned before, we need a procedure for checking whether or not a given symbolic state denotes at least one nonsymbolic global state for which a given terminal predicate is false. Our solution transforms a terminal predicate and a symbolic global state into a set of additional constraints. When the additional constraints are merged with those of the symbolic global state we get a new symbolic state. Every nonsymbolic state denoted by that symbolic state is a state for which the predicate does not hold (and that was denoted by the original symbolic state also).

5.2 Bases

Definition 5.1

$$\mathbf{AC}_{\mathscr{M}}(M) =_{\text{def}} M^{*[\mathscr{M}]} \tag{5.1}$$
$$\mathbf{TC}(M) =_{\text{def}} M^{*[\mathbf{F}(\mathcal{S})]} \tag{5.2}$$
$$\mathbf{NFTC}(M) =_{\text{def}} \mathbf{TC}(M) \cap \mathscr{M} \quad \square \tag{5.3}$$

Thus, $\mathbf{AC}_{\mathscr{M}}(M)$ denotes the <u>a</u>lgebraic <u>c</u>losure of M under $\mathscr{M}$, $\mathbf{TC}(M)$ denotes the free <u>t</u>erm <u>c</u>losure of M, and $\mathbf{NFTC}(M)$ denotes the <u>n</u>ormal <u>f</u>orm <u>t</u>erm <u>c</u>losure of M, i.e. the set of normal-form terms in $\mathbf{TC}(M)$. We use these mnemonic notations from now on.

Lemma 5.1 *For every $M \subseteq \mathscr{M}$, $\mathbf{NFTC}(M) \subseteq \mathbf{AC}_{\mathscr{M}}(M)$ and $\mathbf{NFTC}(M) \subseteq \mathbf{TC}(M)$. $\mathbf{NFTC}$ and $\mathbf{TC}$ are closure operators over $\mathbf{F}(\mathcal{S})$ and $\mathbf{AC}_{\mathscr{M}}$ is a closure operator over $\mathscr{M}$.*

Proof easy; omitted. □

Definition 5.2 Let M be a set of terms. An M-pseudo-free substitution σ is a ground substitution that maps all variables appearing in M to distinct elements of $\mathcal{A}$ such that if v and v' appear in M, then $\sigma(v) = \sigma(v')$ iff $v = v'$, and neither $\sigma(v)$ nor $\sigma(v')$ appear in M. □

Pseudo-free substitutions are mainly a way to handle terms containing variables actually in the domain of ground terms. This eases exposition.

Definition 5.3 A set $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ is *normalizable* iff there exists a substitution σ such that

$$\sigma(M) \subseteq \mathscr{M}. \tag{5.4}$$

Any such σ is a *normalizing substitution* of M. The set of all the normalizing substitutions of M is denoted by $\mathbf{norm}(M)$. □

Definition 5.4 A *basis* [*sub-basis*] of a finite $M \subseteq \mathscr{M}$ is a finite set $M_B \subseteq \mathscr{M}$ such that $\mathbf{NFTC}(M_B) = \mathbf{AC}_{\mathscr{M}}(M)$ [$\mathbf{NFTC}(M_B) \subseteq \mathbf{AC}_{\mathscr{M}}(M)$].

For a set $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$, a *quasi-basis* of M is a set $M_Q \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ such that for all $\sigma \in \mathbf{norm}(M)$ it holds that $\mathbf{NFTC}(\sigma(M_Q)) \subseteq \mathbf{AC}_{\mathscr{M}}(\sigma(M))$.

A quasi-basis M_Q is a *complete* quasi-basis of M iff for all $\sigma \in \mathbf{norm}(M)$, $\mathbf{AC}_{\mathscr{M}}(\sigma(M_Q)) = \mathbf{AC}_{\mathscr{M}}(\sigma(M))$.

A quasi-basis M_Q is a *full* quasi-basis of M iff for all M-pseudo-free substitutions σ, $\sigma(M_Q)$ is a basis of $\sigma(M)$. □

The definition of basis is motivated by the following idea: Assume that we have a set $M \subseteq \mathscr{M}$ and we would like to know whether or not $m \in \mathbf{AC}_{\mathscr{M}}(M)$ for some m, i.e. whether or not $M \vdash m$. Asking this is equivalent to asking whether or not there exists a term t that is an arbitrary composition of the terms in M such that $t \underset{\mathfrak{R}}{\to}^* m$. Checking for this effectively is a problem because the rewrite system is involved.

However, a question $m \overset{?}{\in} \mathbf{NFTC}(M)$ is much easier, because now the rewrite system is not required. Assuming that checking for set inclusion (whether $a \overset{?}{\in} S$ for an S of a reasonable size) can be done in constant time, a simple algorithm can answer this problem in time $O(|m|)$, where $|m|$ is the length of m (actually, $O(n)$ where n is the number of subterms in m, but $n = O(|m|)$).

Therefore, if M_B is a basis of M, the possibly nasty problem $m \overset{?}{\in} \mathbf{AC}_{\mathscr{M}}(M)$ can be reduced to the problem $m \overset{?}{\in} \mathbf{NFTC}(M_B)$.

This is nice. However, the real utility comes forth when M can include terms containing variables, i.e. $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$. In that case the question is to find all substitutions σ such that $\sigma(m) \in \mathbf{AC}_{\mathscr{M}}(\sigma(M))$. This can be even harder than the corresponding problem without variables. However, if M_Q is, say, a full and complete quasi-basis of M, then a simple recursively descending unification algorithm works. This is what is actually done later on.

Another way to see the rôle of M_B, a basis of M, is to think of M_B as an algorithmically efficient representation of M^*, the infinite closure of the terms in M.

Example 5.2 Assume $P_{\text{symenc}}(\mathsf{enc}, \mathsf{dec}) \in \mathscr{D}$. Let $M = \{\mathsf{enc}(K, a), b\}$, where $K \in \mathscr{X}$ and $a, b \in \mathcal{A}$. M is a complete quasi-basis of itself. For every substitution $\sigma = \{K \mapsto x\}$ where $x \notin \{a, b\}$, $\sigma(M)$ becomes a proper basis of itself. Thus M is a full quasi-basis of itself also. On the other hand, the smallest basis of $\{\mathsf{enc}(b, a), b\}$ is $\{a, b\}$, and thus for the substitution $\sigma = \{K \mapsto b\}$, $\sigma(M)$ is only a sub-basis of itself ($\mathbf{NFTC}(\sigma(M))$ does not contain the term a).

Lemma 5.3 *Let* $A, B \subseteq \mathscr{M}$ *such that* $\mathbf{NFTC}(A) = \mathbf{NFTC}(B)$. *Then* $\mathbf{NFTC}(A \cap B) = \mathbf{NFTC}(A) = \mathbf{NFTC}(B)$.

Proof. The proof is by structural induction over $\mathbf{T}_{\mathcal{S}}(\emptyset)$. As the base case, let $t \in \mathcal{A}$. Obviously, $t \in \mathbf{NFTC}(A)$ iff $t \in A$; similarly for $t \in \mathbf{NFTC}(B)$. Therefore $t \in \mathbf{NFTC}(A) \Leftrightarrow t \in \mathbf{NFTC}(B)$ because $\mathbf{NFTC}(A) = \mathbf{NFTC}(B)$ and thus $t \in A \Leftrightarrow t \in B$. Therefore $t \in \mathbf{NFTC}(A)$ iff $t \in \mathbf{NFTC}(B)$ iff $t \in \mathbf{NFTC}(A \cap B)$.

Assume then that for $t_1, \ldots, t_k$, $t_i \in \mathbf{NFTC}(A \cap B)$ iff $t_i \in \mathbf{NFTC}(A)$ iff $t \in \mathbf{NFTC}(B)$. Consider $t = f(t_1, \ldots, t_k)$ such that t is in its normal form. There are some cases.

(1) $t_1, \ldots, t_k \in \mathbf{NFTC}(A)$. By the induction hypothesis, then $t_1, \ldots, t_k \in \mathbf{NFTC}(B)$ and $t_1, \ldots, t_k \in \mathbf{NFTC}(A \cap B)$. Thus, $t \in \mathbf{NFTC}(A)$, $t \in \mathbf{NFTC}(B)$ and $t \in \mathbf{NFTC}(A \cap B)$. **(2)** There exists t_i such that $t_i \notin \mathbf{NFTC}(A)$, but $t \in \mathbf{NFTC}(A)$. Then $t \in A$. By the induction hypothesis, $t_i \notin \mathbf{NFTC}(B)$. Therefore, because $\mathbf{NFTC}(A) = \mathbf{NFTC}(B)$, $t \in B$. Thus, $t \in A \cap B$. Therefore $t \in \mathbf{NFTC}(A \cap B)$. **(3)** $t \notin \mathbf{NFTC}(A)$. Then there must be t_i such that $t_i \notin \mathbf{NFTC}(A)$, and moreover, $t \notin A$. By the induction hypothesis, $t_i \notin \mathbf{NFTC}(B)$ and $t_i \notin \mathbf{NFTC}(A \cap B)$. Also, $t \notin B$ because $\mathbf{NFTC}(A) = \mathbf{NFTC}(B)$. Trivially, $t \notin A \cap B$. Thus $t \notin \mathbf{NFTC}(A \cap B)$. This completes the induction and the proof. □

Lemma 5.4 *Let* M *be a finite subset of* $\mathscr{M}$. *Then* M *has a basis.*

Proof. Let S denote the set of all the subterms of the terms in M, including M. Formally, S is the smallest set for which **(1)** $M \subseteq S$, and **(2)** $f(t_1, \ldots, t_k) \in S \Rightarrow t_1, \ldots, t_k \in S$. Let B denote $\mathbf{AC}_{\mathscr{M}}(M) \cap S$.

We assume now for a while the that the signature verification rewrite rule does not exist (recall the definition of the rewrite system, shown in Figure 4.1). With this assumption, if $t \underset{\mathfrak{R}\,t}{\rightarrow} t'$ then t' must be a subterm of t.

Let now x be chosen arbitrarily so that $x \in \mathbf{AC}_{\mathscr{M}}(M)$. Then there exist a sequence of sets $B_0, \ldots, B_n$ so that $B_0 = B$ and $x \in B_n$, and where $B_i = B_{i-1} \cup \{\mathfrak{N}(f(t_1, \ldots, t_\ell))\}$ for some $t_1, \ldots, t_\ell \in B_{i-1}$. (This is by the definition of $\mathbf{AC}_{\mathscr{M}}(M)$, algebraic closure under $\mathscr{M}$.)

We now show that for all $i \in \{0, \ldots, n\}$, $B_i \subseteq \mathbf{NFTC}(B)$. This leads to the conclusion that $x \in \mathbf{NFTC}(B)$. Because x was chosen arbitrarily, B would be a basis of M if the signature verification rewrite rule did not exist. However, the addition of the rewrite rule only requires us to either add the term TRUE or no terms to B to make B really a basis of M. This is because applying the rewrite rule can result only in TRUE.

Obviously $B_0 \subseteq \mathbf{NFTC}(B)$ because $B = B_0$ and B contains only normal-form terms. We proceed by mathematical induction over i.

The induction hypothesis is: If $t \in B_i$ and t' is a subterm of t so that t' belongs to $\mathbf{AC}_{\mathscr{M}}(M)$, then $t' \in \mathbf{NFTC}(B_i)$. Also, $\mathbf{NFTC}(B_i) = \mathbf{NFTC}(B)$.

For $i = 0$ this holds by the definition of B.

Let us now assume the hypothesis holds for i. Assume $B_{i+1} \setminus B_i$ is nonempty (it might be empty, but in this case the induction step is trivial). Then there exists exactly one term t such that $t \in B_{i+1} \setminus B$. Because of the way how B_{i+1} is constructed, t is obtained from a $t' \in \mathbf{TC}(B_i)$ by applying the rewrite system *at most once on the top level.*

Assume first $t = t'$. Then all the immediate subterms of t belong to $\mathbf{NFTC}(B_i)$. Therefore if $t'' \in B_{i+1}$ and t''' is a subterm of to t'' that belongs to $\mathbf{AC}_{\mathscr{M}}(M)$, $t''' \in \mathbf{NFTC}(B_{i+1})$ by the induction hypothesis. Also, because $t \in \mathbf{NFTC}(B_i)$, $\mathbf{NFTC}(B_{i+1}) = \mathbf{NFTC}(B_i)$. Thus the induction hypothesis holds also for $i+1$.

Assume then $t' \underset{\mathfrak{R}\,t}{\rightarrow} t$. Observe that t must be a subterm of t'. This is because of the structure of the rewrite system and the assumption that the signature verification rewrite rules does not exist. t must belong to $\mathbf{AC}_{\mathscr{M}}(M)$. Because t is a subterm of t', t is a subterm of some term belonging to B_i. Therefore, by induction hypothesis, $t \in \mathbf{NFTC}(B_i)$. Therefore, $\mathbf{NFTC}(B_{i+1}) = \mathbf{NFTC}(B_i)$. Thus, by the induction hypothesis, $\mathbf{NFTC}(B_{i+1}) = \mathbf{NFTC}(B)$.

To show that the other part of the induction hypothesis also holds for $i + 1$, observe that the set of the subterms of terms in B_{i+1} is equal to that of B_i, because the added term was a subterm of some terms in B_i.

Thus the hypothesis holds for all i by induction. In particular, it holds for $i = n$. Because $\mathbf{NFTC}(B_n) = \mathbf{NFTC}(B)$, it must be that $B_n \subseteq \mathbf{NFTC}(B)$. Because $x \in B_n$, $x \in \mathbf{NFTC}(B)$.

B is thus either a basis of M, or it can be made one by adding TRUE to it. This concludes the proof of the existence of a basis. □

Lemma 5.5 *Let M be a finite subset of $\mathscr{M}$. All the bases of M form a non-empty partial order with $\subseteq$ that has a bottom element.*

Proof. M has at least one basis by Lemma 5.4. Let then A, B be bases of M. By the definition of a basis, $\mathbf{NFTC}(A) = \mathbf{NFTC}(B)$. By Lemma 5.3 $\mathbf{NFTC}(A \cap B) = \mathbf{NFTC}(A)$. Therefore $A \cap B$ is also a basis of M. Thus in particular

$$\bigcap_{B \text{ is a basis of } M} B \tag{5.5}$$

is a basis of M, and this basis must be a subset of all the bases. □

Corollary 5.6 *Every finite subset of $\mathscr{M}$ has a unique smallest basis.* □

Lemma 5.7 *Let $\varphi : \mathcal{A} \mapsto \mathcal{A}$ be a possibly non-bijective mapping from $\mathcal{A}$ to the same set. Let M be a finite subset of $\mathscr{M}$. Let B be a basis of M. Let us extend φ implicitly to arbitrary terms in the natural way. Assume that $\varphi(B) \in \mathscr{M}$ (some choices of φ can cause terms not in their normal forms to appear in $\varphi(B)$). Then $\mathbf{NFTC}(\varphi(B)) \subseteq \mathbf{AC}_{\mathscr{M}}(\varphi(M))$, i.e. $\varphi(B)$ is a sub-basis of $\varphi(M)$.*

Proof.

$$\begin{aligned}\mathbf{NFTC}(\varphi(B)) = \varphi(\mathbf{NFTC}(B)) = \varphi(\mathbf{AC}_{\mathscr{M}}(M)) &\subseteq \mathbf{AC}_{\mathscr{M}}(\varphi(\mathbf{AC}_{\mathscr{M}}(M))) \\ &= \mathbf{AC}_{\mathscr{M}}(\mathbf{AC}_{\mathscr{M}}(\varphi(M))) = \mathbf{AC}_{\mathscr{M}}(\varphi(M)). \end{aligned} \tag{5.6}$$

The first equality comes from the fact that the rewrite relation is not involved for **NFTC**. The second equality holds because $\mathbf{NFTC}(B) = \mathbf{AC}_{\mathscr{M}}(M)$ as B is a basis of M. The subset relation holds because $\mathbf{AC}_{\mathscr{M}}$ is a closure operator over $\mathscr{M}$. Next, $\varphi(\mathbf{AC}_{\mathscr{M}}(M)) \subseteq \mathbf{AC}_{\mathscr{M}}(\varphi(M))$ because if $m \underset{\mathfrak{R}}{\to}^* m'$ then $\varphi(m) \underset{\mathfrak{R}}{\to}^* \varphi(m')$. Thus if $m \in \mathbf{AC}_{\mathscr{M}}(M)$, then $\varphi(m) \in \mathbf{AC}_{\mathscr{M}}(\varphi(M))$. Because $\varphi(M) \subseteq \varphi(\mathbf{AC}_{\mathscr{M}}(M)) \subseteq \mathbf{AC}_{\mathscr{M}}(\varphi(M))$, $\mathbf{AC}_{\mathscr{M}}(\varphi(\mathbf{AC}_{\mathscr{M}}(M))) = \mathbf{AC}_{\mathscr{M}}(\mathbf{AC}_{\mathscr{M}}(\varphi(M)))$ because $\mathbf{AC}_{\mathscr{M}}$ is a closure operator. The last equality stems from the same fact. □

Lemma 5.8 *Let $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ be finite. Let π be an M-pseudo-free substitution and π^{-1} its inverse, which exists by the definition of a pseudo-free substitution. Let B be a basis of $\pi(M)$. Then $\pi^{-1}(B)$ is a complete and full quasi-basis of M.*

Proof. Let $\sigma \in \mathbf{norm}(M)$. We can represent σ as $\varphi \circ \pi$ where φ is a suitably chosen function $\mathcal{A} \mapsto \mathcal{A}$. Because σ is a normalizing substitution of M, φ can be chosen so that the conditions of Lemma 5.7 hold. Therefore

$$\begin{aligned}\mathbf{NFTC}(\sigma(\pi^{-1}(B))) &= \mathbf{NFTC}(\varphi(\pi(\pi^{-1}(B)))) \\ &= \mathbf{NFTC}(\varphi(B)) \subseteq \mathbf{AC}_{\mathscr{M}}(\varphi(\pi(M))) = \mathbf{AC}_{\mathscr{M}}(\sigma(M)). \quad (5.7)\end{aligned}$$

Thus $\pi^{-1}(B)$ is quasi-basis of M.

Obviously, if $\sigma = \pi$, i.e. σ is the single pseudo-free substitution chosen, then $\sigma(\pi^{-1}(B))$ is a basis of $\sigma(M)$. M-pseudo-free substitutions can be seen however to be all equivalent because the choice of the distinct atoms variables are mapped to is not significant. This stems from the structure of the rewrite system; the rewrite system does not mention any constants (except for TRUE, but it does not appear on the left-hand sides which is enough). Therefore $\sigma(\pi^{-1}(B))$ is a basis of $\sigma(M)$ for *all* the M-pseudo-free substitutions σ. Therefore $\pi^{-1}(B)$ is a full quasi-basis of M.

Finally, assume $\sigma \in \mathbf{norm}(M)$. Let again $\sigma = \varphi \circ \pi$ for some $\varphi : \mathcal{A} \mapsto \mathcal{A}$. Then

$$\begin{aligned}\mathbf{AC}_{\mathscr{M}}(\sigma(\pi^{-1}(B))) &= \mathbf{AC}_{\mathscr{M}}(\varphi(\pi(\pi^{-1}(B)))) = \mathbf{AC}_{\mathscr{M}}(\varphi(B)) \\ &= \mathbf{AC}_{\mathscr{M}}(\varphi(\mathbf{AC}_{\mathscr{M}}(B))) = \mathbf{AC}_{\mathscr{M}}(\varphi(\mathbf{AC}_{\mathscr{M}}(\pi(M)))) \\ &= \mathbf{AC}_{\mathscr{M}}(\varphi(\pi(M))) = \mathbf{AC}_{\mathscr{M}}(\sigma(M)). \quad (5.8)\end{aligned}$$

Therefore $\pi^{-1}(B)$ is a complete and full quasi-basis of M, as we just showed completeness. □

Lemma 5.9 *Let $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ be finite and let Q be a full quasi-basis of M. Then Q can be always obtained as in the previous lemma.*

Proof. Because Q is a full quasi-basis of M, let π be any M-pseudo-free substitution. Then $\pi(Q)$ is a basis of $\pi(M)$ by definition. □

Lemma 5.10 *Let $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ be finite and let π be a fixed M-pseudo-free substitution. Then there exists a bijection β from the set of the complete and full quasi-bases of M to the set of bases of $\pi(M)$ such that*

$$Q_1 \subseteq Q_2 \Leftrightarrow \beta(Q_1) \subseteq \beta(Q_2). \quad (5.9)$$

Proof. Every full quasi-basis of M is $\pi^{-1}(B)$ for some basis B of $\pi(M)$ by Lemma 5.9. Similarly, every basis B of $\pi(M)$ corresponds to the complete and

```
function IN-CLOSURE?(m, M)
if m ∈ M then
    return true
else if m = f(m_1, ..., m_k) then
    for i = 1 to k do
        if ¬IN-CLOSURE?(m_i, M) then
            return false
        end if
    end for
    return true
else
    return false
end if
```

Figure 5.6 Function IN-CLOSURE?.

full quasi-basis $\pi^{-1}(B)$ of M by Lemma 5.7. This gives the bijection

$$\beta = \pi. \tag{5.10}$$

The result now follows easily. □

Corollary 5.11 *Every full quasi-basis is also complete.* □

Lemma 5.12 *Every finite set $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ has a unique smallest complete and full quasi-basis.*

Proof. This result now follows from Corollary 5.6 and Lemma 5.10. □

The importance the lemmata above is in that now no special algorithm is needed for computing quasi-bases. It is enough to compute the basis of a ground term set. Then a quasi-basis of the corresponding non-ground term set is obtained straightforwardly.

The function QUASI-BASIS, presented in Figure 5.7, computes the unique, smallest, full and complete quasi-basis of a set of terms $M = \{m_1, \ldots, m_k\}$ ($M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$). This fact is formalized in Theorem 5.14. However, an auxiliary procedure is needed first.

Lemma 5.13 IN-CLOSURE?(m, M) *(see Figure 5.6) returns 'true' iff $\sigma(m) \in$ $\mathbf{TC}(\sigma(M))$ for all substitutions σ. (M is assumed to be finite.)*

Proof. Clearly IN-CLOSURE? terminates. Let φ be below an $(\{m\} \cup M)$-pseudo-free substitution. We prove the claim by structural induction over the arguments m.

function QUASI-BASIS$(M = \{m_1, \dots, m_k\})$
for $i = 1$ to k **do**
  $R \leftarrow \{m_1, \dots, m_{i-1}, m_{i+1}, \dots, m_k\}$
  if IN-CLOSURE?(m_i, R) **then** *; m_i derivable*
    return QUASI-BASIS(R) *; case a*
  else if $m_i = f(t_1, \dots, t_n)$ **then** *; otherwise m_i must be compound*
    if $P_{n\text{-tuple}}(f, \dots) \in \mathscr{D}$ **then**
      return QUASI-BASIS$(R \cup \{t_1, \dots, t_n\})$ *; case b*
    else if $n = 1$ **then**
      if $P_{\text{inv}}(f', f) \in \mathscr{D}$ **then**
        return QUASI-BASIS$(R \cup \{t_1\})$ *; case c*
      end if
    else if $n = 2$ **then**
      if $P_{\text{symenc}}(f, d) \in \mathscr{D} \wedge$ IN-CLOSURE?(t_1, R) **then**
        return QUASI-BASIS$(R \cup \{t_2\})$ *; case d*
      else if $P_{\text{pubenc}}(f, d, p, s) \in \mathscr{D} \wedge t_1 = p(k) \wedge$ IN-CLOSURE?$(s(k), R) \wedge$ $\neg$IN-CLOSURE?(t_2, R) **then**
        return QUASI-BASIS$(R \cup \{m_i, t_2\})$ *; case e*
      end if
    end if
  end if
end for
return $\{m_1, \dots, m_k\}$ *; case f*

Figure 5.7 Function QUASI-BASIS.

Line 1: If $m \in M$ then obviously $\sigma(m) \in \sigma(M) \subseteq \mathbf{TC}(\sigma(m))$ for all substitutions σ.

Line 3: If $m \in \mathcal{A} \cup \mathcal{X}$ and $m \notin M$ then $\varphi(m) \notin \mathbf{TC}(\varphi(M))$.

Lines 4–9: Assume $f = (m_1, \dots, m_k)$ and that IN-CLOSURE? gives the correct answer for all subterms m_i. If IN-CLOSURE? returns 'true' for all the subterms, then for all σ and m_i, $\sigma(m_i) \in \mathbf{TC}(\sigma(M))$. Therefore also $f(\sigma(m_1), \dots, \sigma(m_k)) \in \mathbf{TC}(\sigma(M))$. On the other hand, assume IN-CLOSURE?(m_i) returns 'false'. Then $\varphi(m_i) \notin \mathbf{TC}(\varphi(M))$. Therefore necessarily should be $\varphi(m) \in \varphi(M)$ but this implied $m \in M$ which cannot hold because the algorithm did not stop on the line 2. □

Theorem 5.14 QUASI-BASIS(M) *returns the smallest full and complete quasi-basis of M for any normalizable $M \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ such that* TRUE $\in M$.

Proof. By the earlier observations, it is enough to show that QUASI-BASIS(M) returns the unique smallest basis of M whenever $M \subseteq \mathscr{M}$. (The algorithm could have been augmented with an explicit pseudo-free substitution and its inverse. They can be however dispensed with because the algorithm can handle variables in terms directly in the same way as atomic terms are handled.)

The algorithm is a tail-recursive iterative process that modifies a given set M on every iteration.

Actually, QUASI-BASIS is an interleaving of two processes. The first process adds to M all subterms of terms in M that belong to $\mathbf{AC}_{\mathscr{M}}(M)$. The second one removes then terms from M so long as $\mathbf{NFTC}(M)$ remains constant. The first process ensures that eventually M is a basis of itself (see the Proof of Lemma 5.4); the second process traverses the partial order of the bases of M downwards until the bottom element is reached.

The exact details are relatively straightforward. They are left to the reader. □

5.3 Symbolic States

Let $\mathcal{X}$ be a set of variables as before. Our symbolic representation of sets of global states is as follows.

Definition 5.5 A symbolic local state is a triple $\langle p, \hat{\Upsilon}, c\rangle$ where $p \in \mathcal{P}$, $\hat{\Upsilon} \in \mathbf{subst}(\mathcal{V}, \mathbf{T}_{\mathcal{S}}(\mathcal{X}))$ and $c \in \mathbb{N}$. The set of all the symbolic local states is denoted by $\hat{\mathcal{L}}$. □

Definition 5.6 A symbolic global quasi-state is a triple $\langle \hat{\mathscr{L}}, M, C\rangle$ where $M \subset \mathbf{T}_{\mathcal{S}}(\mathcal{X})$, M finite, $\hat{\mathscr{L}}$ is a total function from $\mathcal{N}$ to $\hat{\mathcal{L}}$ and C is a set of constraints to be defined later. The set of symbolic global quasi-states is denoted by $\hat{\mathcal{G}}_*$. □

Definition 5.7 We define the representation function $\rho : \hat{\mathcal{G}}_* \mapsto 2^{\mathcal{G}}$ as follows:

$$\rho(\langle \hat{\mathscr{L}}, M, C\rangle) = \{\langle \sigma(\hat{\mathscr{L}}), \sigma(M)\rangle \mid \begin{array}{l} \sigma \in \mathbf{subst}(\mathcal{X}, \mathscr{M}), \\ \sigma(C) \text{ is non-contradictory.}\} \end{array} \tag{5.11}$$

Here $\sigma(\hat{\mathscr{L}})$ is defined as

$$\left[\hat{\mathscr{L}}(n) = \langle p, \hat{\Upsilon}, c\rangle \wedge \hat{\Upsilon}(v) = m\right] \Rightarrow \left[\sigma(\hat{\mathscr{L}}(n)) = \langle p, \Upsilon', c\rangle \wedge \Upsilon'(v) = \sigma(m)\right]. \tag{5.12}$$

Any such σ is said to be a *realizing substitution* for the corresponding symbolic global quasi-state. The set of the most general realizing substitutions for a symbolic global quasi-state $\hat{g}$ is denoted by $\mathbf{real}(\hat{g})$, i.e. the smallest set such that

$$\sigma \text{ is a realising substitution of } \hat{g} \tag{5.13}$$

implies

$$(\exists \sigma' \in \mathbf{real}(\hat{g}))[\sigma' \trianglelefteq \sigma]. \tag{5.14}$$

Of course, all realizing substitutions must be grounding for $\hat{g}$. This definitions gets rid of the ambiguity that stems from the fact that the substitution could mention variables that do not appear at all in $\hat{g}$. □

Definition 5.8 The set of *symbolic global states* is the largest subset of $\hat{\mathcal{G}}_*$ such that if $\hat{g} = \langle \hat{\mathscr{L}}, M, C\rangle$ belongs to this subset, then for all $\sigma \in \mathbf{real}(\hat{g})$ it holds that $\sigma(\hat{\mathscr{L}}_2)(n)(v)$ is in its normal form for all n, v, i.e. belongs to $\mathscr{M}$, and $\sigma(M) \subset \mathscr{M}$. The set of symbolic global states is denoted by $\hat{\mathcal{G}}$. □

This definition is in a sense bad because it gives global states as objects defined via a semantical definition instead of an easily understandable syntactic one. However, we will guarantee in the development later that all the symbolic global states encountered during analysis indeed fulfill the conditions of Definition 5.8. On the other hand, the conditions of Definition 5.8 will also be *assumed*. This is actually big induction that will be taken care of during the development.

5.3.1 Constraints

The purpose of constraint sets is to restrict the set of acceptable substitutions σ in (5.11). Here we define what constraint sets are and when they are contradictory.

Definition 5.9 Let $\mathscr{K} = \{*_1, *_2, \dots\}$ be a set of *pattern variables*. Then the following are *constraints* for all $a, b, m \in \mathbf{T}_\mathcal{S}(\mathcal{X})$, $p \in \mathbf{T}_\mathcal{S}(\mathcal{X} \cup \mathscr{K})$, $n \in \mathbb{Z}_+$, and $M \subset \mathbf{T}_\mathcal{S}(\mathcal{X})$, M finite:

1. $a \overset{\mathbf{c}}{\doteq} b$,
2. $a \overset{\mathbf{c}}{\not\approx} p$,
3. $[n] \overset{\mathbf{c}}{\vdash} m$, and
4. $[n] \overset{\mathbf{c}}{\doteq} [n-1] \cup M$.

A *constraint set* is a finite set $\{c_1, \dots, c_k\}$ of constraints such that for every n, at most one constraint of the form $[n] \overset{\mathbf{c}}{\doteq} [n-1] \cup M$ is in the set, and if $[n] \overset{\mathbf{c}}{\doteq} [n-1] \cup M \in C$ and $n > 1$, then also $[n-1] \overset{\mathbf{c}}{\doteq} [n-2] \cup M' \in C$ for some M'. The set of all constraint sets is denoted by $\mathfrak{C}$. □

Definition 5.10 Given a constraint set $C \in \mathfrak{C}$, we define the function $[\cdot] : \mathbb{N} \mapsto 2^{\mathbf{T}_\mathcal{S}(\mathcal{X})}$ as follows:

$$[n] = \begin{cases} \{\text{TRUE}\} & \text{if } n = 0 \\ [n-1] \cup M & \text{if } [n] \overset{\mathbf{c}}{\doteq} [n-1] \cup M \text{ is in } C \\ \bot & \text{otherwise.} \end{cases} \tag{5.15}$$

This definition assumes that C is understood from the context. □

Definition 5.11 We define the predicate match over $\mathbf{T}_\mathcal{S}(\emptyset) \times \mathbf{T}_\mathcal{S}(\mathscr{K})$ as follows:

$$\text{match}(t, p) \Leftrightarrow (\exists \sigma \in \mathbf{subst}(\mathscr{K}, \mathbf{T}_\mathcal{S}(\emptyset)))[t = \sigma(p)]. \qquad \square \tag{5.16}$$

Definition 5.12 A ground constraint set is a constraint set that does not contain any occurrences of variables, i.e. elements of $\mathcal{X}$. A ground constraint set is *non-contradictory* iff for every element $c \in C$,

1. if $c = a \overset{\mathbf{c}}{\doteq} b$, then $a, b \in \mathscr{M}$ and $a = b$;

2. if $c = a \overset{\mathbf{c}}{\not\approx} b$, then $a \in \mathscr{M}$, and $\neg\text{match}(a, b)$;

3. if $c = [n] \overset{\mathbf{c}}{\vdash} m$, then $[n] \neq \bot$, $m \in \mathbf{AC}_{\mathscr{M}}([n])$, $m \in \mathscr{M}$ and $[n] \subseteq \mathscr{M}$. $\square$

Definition 5.13 A *solution* to a constraint set C is a ground substitution σ such that the ground set $\sigma(C)$ is non-contradictory. If C has no solutions, it is *unsatisfiable*. The set of all most general solutions of C is denoted by $\mathbf{sol}(C)$. $\square$

Observe that if $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$, then $\mathbf{real}(\hat{g}) = \mathbf{sol}(C)$ because of Definition 5.8.

Definition 5.14 A constraint set C is in the *simple normal form* if the following hold:

1. There are no constraints of the form $a \overset{\mathbf{c}}{\doteq} b$ in C.

2. If $[n] \overset{\mathbf{c}}{\vdash} m$ belongs to C, then $m \in \mathcal{X}$.

The subset of $\mathfrak{C}$ containing exactly the constraint sets in the simple normal form is denoted by $\mathfrak{C}_s$. $\square$

The intuitive meaning of constraint sets is as follows: A constraint $a \overset{\mathbf{c}}{\doteq} b$ constraints a and b to be the same message. $a \overset{\mathbf{c}}{\not\approx} b$ constraints a and b to be distinct messages, with the extension that b may contain 'wild cards' (elements of $\mathscr{K}$) that are matched by any submessage in a at the same position. $[n] \overset{\mathbf{c}}{\doteq} [n-1] \cup M$ is not a real constraint but is used to specify that the message set $[n]$ is the same as the union of $[n-1]$ and M. $[n] \overset{\mathbf{c}}{\vdash} m$ is the central type of constraints; it constraints m to those terms derivable from the set $[n]$, i.e. those that belong to its algebraic closure.

5.4 Constraint Solver

As the symbolic local states are defined via the solutions of constraint sets, we need a procedure for computing these solutions. Unfortunately, the set of solutions can be infinite. However, much as in the case of unification, we can compute in a sense a set of most general solutions that encompasses all the rest.

Let $\hat{g}, \hat{g}_1, \ldots, \hat{g}_k \in \hat{\mathcal{G}}$ be symbolic global states such that

$$\rho(\hat{g}) = \bigcup_{i=1}^{k} \rho(\hat{g}_i), \tag{5.17}$$

i.e. so that the states represented by $\hat{g}$ are exactly those represented by at least one element of the set $\{\hat{g}_1, \ldots, \hat{g}_k\}$. For the purposes of the analysis, it seems that we can replace $\hat{g}$ with the set $\hat{g}_1, \ldots, \hat{g}_k$, and vice versa. This motivates the following definition.

Definition 5.15 Let Z be a function $\hat{\mathcal{G}} \mapsto 2^{\hat{\mathcal{G}}}$ such that for all $\hat{g} \in \hat{\mathcal{G}}$, $|\rho(\hat{g})| \in \mathbb{N}$

$$\rho(\hat{g}) = \bigcup_{\hat{g}' \in Z(\hat{g})} \rho(\hat{g}'). \tag{5.18}$$

Then Z is said to be an *admissible symbolic state one-to-many transformation.* □

In the following, we develop a particular admissible symbolic state one-to-many transformation REDUCE with the following extra properties:

(1) If $\hat{g}' \in \text{REDUCE}(\hat{g})$ and $\hat{g}' = \langle \hat{\mathscr{L}}, M, C \rangle$, then $C \in \mathfrak{C}_s$, and

(2) if $\rho(\hat{g}') = \emptyset$, then $\hat{g}' \notin \text{REDUCE}(\hat{g})$ for any $\hat{g} \in \hat{\mathcal{G}}$.

That these properties can exist is proven directly by the construction of REDUCE later in this chapter.

The function REDUCE to be developed is, in essence, a constraint solver, that 'solves' possibly complex constraint sets by reducing them to a set of 'simple' constraint sets (namely, sets in the simple normal form). Moreover — which is important — it detects unsatisfiable sets (those C for which $\mathbf{sol}(C) = \emptyset$) and discards them. This is important, because then it holds that

$$\text{REDUCE}(\hat{g}) = \emptyset \text{ iff } \rho(\hat{g}) = \emptyset. \tag{5.19}$$

Thus REDUCE can be used to detect void symbolic global states that need not to be considered any more.

5.4.1 Reduction Loop

Recall that our purpose is to create an admissible symbolic state one-to-many transformation as defined in Definition 5.15.

function REDUCE($\hat{g} \in \hat{\mathcal{G}}$)
$R \leftarrow \{\hat{g}\}$
$F \leftarrow \emptyset$
while $R \neq \emptyset$ **do**
  Choose $\hat{g}' \in R$
  $R \leftarrow R \setminus \{\hat{g}'\}$
  $Q \leftarrow$ REDUCE-STEP($\hat{g}'$)
  if $Q = \{\hat{g}'\}$ **then**
    $F \leftarrow F \cup Q$
  else
    $R \leftarrow R \cup Q$
  end if
end while
return F

Figure 5.8 Basic structure of the constraint solver.

Our particular transformation REDUCE is obtained by iterating a simple transformation named REDUCE-STEP until a fixpoint has been found. The basic schematic structure is presented in Figure 5.8. Quite obviously REDUCE is an admissible symbolic state one-to-many transformation if **(1)** REDUCE-STEP is and **(2)** the algorithm presented in Figure 5.8 terminates, i.e. a fixpoint of REDUCE-STEP is eventually reached on every branch of the computation. In order for REDUCE to fulfill the two extra requirements mentioned at the beginning of this section, it must hold that

(R1) REDUCE-STEP($\hat{g}$) $= \{\hat{g}\}$ implies that if $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$ then $C \in \mathfrak{C}_s$, and that

(R2) if $\rho(\hat{g}) = \emptyset$, then REDUCE-STEP($\hat{g}$) $\neq \{\hat{g}\}$.

We will next present an algorithm that computes the function REDUCE-STEP and then discuss its correctness.

Definition of REDUCE-STEP

The function REDUCE-STEP is defined algorithmically in Figure 5.9. For the purposes of the algorithm, a new type of 'constraint' must be defined. If $n \in \mathbb{Z}_+$, then $\overset{\mathbf{c}}{B}(n)$ is a constraint and thus can be included in a constraint set. However, the semantics of $\overset{\mathbf{c}}{B}(n)$ are not definable in the terms of ground constraint sets. Instead, we define that if $\overset{\mathbf{c}}{B}(n)$ belongs to C, then for all solutions σ of C, it must be that $\sigma(\text{QUASI-BASIS}([k]))$ is a basis of $\sigma([k])$ for all $k \leq n$. This type of 'metaconstraint' is needed only during the computation of REDUCE; its instances can be removed from the constraint set returned by the algorithm, although it is better not to do so because that can save computation time later.

The function EXPAND is defined after the general correctness of REDUCE-STEP is discussed.

Correctness of REDUCE-STEP

Lemma 5.15 REDUCE-STEP *is an admissible symbolic state one-to-many transformation fulfilling R1 and R2.*

Proof. Consider the cases a–h, denoted in the listing of REDUCE-STEP.

a) In this case, there exists a constraint $a \stackrel{\mathbf{c}}{=} b$ in C such that $\text{unify}(a, b) = \mu \neq \perp$. Because for every $\sigma \in \mathbf{sol}(C)$, it must be that $\sigma(a) = \sigma(b)$, then every $\sigma \in \mathbf{sol}(C)$ can be expressed as $\sigma' \circ \mu$ for some σ', i.e. $\mu \trianglelefteq \sigma'$. Therefore, every noncontradictory ground instance of C is a noncontradictory ground instance of $\mu(C)$ and vice-versa. Therefore C can be replaced with $\mu(C)$ and thus $\hat{g}$ with $\mu(\hat{g})$. However, in $\mu(C)$, the constraint $a \stackrel{\mathbf{c}}{=} b$ has reduced to $\mu \stackrel{\mathbf{c}}{=} (a)\mu(b)$ which is $x \stackrel{\mathbf{c}}{=} x$ for some x because μ is the mgu of a and b; therefore this constraint is trivially satisfied and can be removed.

b) In this case, a and b are as above but they do not unify. Therefore, there cannot in particular be any $\sigma \in \mathbf{sol}(C)$ such that $\sigma(a) = \sigma(b)$. Therefore, $\mathbf{sol}(C) = \emptyset$. Thus C is unsatisfiable and $\rho(\hat{g}) = \emptyset$. Thus $\emptyset$ can be returned.

c) In this case, there exists a constraint $a \stackrel{\mathbf{c}}{\not\approx} b$ in C such that a and b cannot be unified when unification is done using both elements of $\mathcal{X}$ and those of $\mathscr{K}$. Thus, if $\sigma \in \mathbf{sol}(C)$, trivially $\sigma(a) \neq \sigma(b)$. Therefore the set of solutions to C does not change when this particular constraint is removed from C.

d) In this case, a matches b for all possible substitutions σ. Therefore, in particular, there cannot be any $\sigma \in \mathbf{sol}(C)$ such that $\sigma(a)$ would not match $\sigma(b)$. Therefore, C is unsatisfiable and $\rho(\hat{g}) = \emptyset$. Thus, $\emptyset$ can be returned.

e) In this case, there exists a constraint $[n] \stackrel{\mathbf{c}}{\vdash} m$ in C such that $\sigma(m) \in \mathbf{TC}(\sigma([n]))$ for all substitutions σ. Therefore, in particular, $\sigma(m) \in \mathbf{TC}(\sigma([n]))$ for all $\sigma \in \mathbf{norm}(m)$. Because every $\sigma \in \mathbf{sol}$ must belong to $\mathbf{norm}(m)$ (by the definition of a noncontradictory constraint set), it must be that $\sigma \in \mathbf{sol} \Rightarrow \sigma(m) \in \mathbf{NFTC}(\sigma([n])) \subseteq \mathbf{AC}_{\mathscr{M}}(\sigma([n]))$. Thus,

function REDUCE-STEP($\hat{g}$)
$\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$
$C = \langle c_1, \ldots, c_k \rangle$
for $i = 1$ to k **do** *; Check equalities and inequalities*
  $C' \leftarrow C \setminus c$
  if $c_i = a \stackrel{\mathbf{c}}{=} b$ **then** *; Equality*
    $\mu \leftarrow \mathrm{unify}(a, b)$
    if $\mu \neq \perp$ **then**
      return $\{\mu(\langle \hat{\mathscr{L}}, M, C' \rangle)\}$ *; case a*
    else
      return $\emptyset$ *; case b*
    end if
  else if $c_i = a \stackrel{\mathbf{c}}{\not\approx} b$ **then** *; Inequality*
    if $(\forall \sigma \in \mathbf{subst}(\mathcal{X}, \mathbf{T}_{\mathcal{S}}(\emptyset)))[\neg \mathrm{match}(\sigma(a), \sigma(b))]$ **then** *; Trivially fulfilled*
      return $\{\langle \hat{\mathscr{L}}, M, C' \rangle\}$ *; case c*
    else if $(\forall \sigma \in \mathbf{subst}(\mathcal{X}, \mathbf{T}_{\mathcal{S}}(\emptyset)))[\mathrm{match}(\sigma(a), \sigma(b))]$ **then** *; Always contradictory*
      return $\emptyset$ *; case d*
    end if
  end if
end for
for $n = 0$ to $\max y : [y] \stackrel{\mathbf{c}}{=} [y-1] \cup M \in C$ **do** *; Derivability constraints*
  for $i = 1$ to k **do** *; Loop over constraints*
    $C' \leftarrow C \setminus c_i$
    if $c_i = [n] \stackrel{\mathbf{c}}{\vdash} m$ **then**
      if INCLOSURE?$(m, [n])$ **then** *; Trivially fulfilled?*
        return $\{\langle \hat{\mathscr{L}}, M, C' \rangle\}$ *; case e*
      else if $m \notin \mathcal{X}$ and $\overset{\mathbf{c}}{B}(q) \in C$ for some $q \geq n$ **then**
        $R \leftarrow \emptyset$ *; Here when $[n]$ is a basis of itself*
        $\{b_1, \ldots, b_m\} \leftarrow$ QUASI-BASIS$([n])$
        for $j = 1$ to m **do**
          if $b_j \notin \mathcal{X}$ **then**
            $\mu \leftarrow \mathrm{unify}(m, b_i)$
            if $\mu \neq \perp$ **then**
              $R \leftarrow R \cup \{\mu(\langle \hat{\mathscr{L}}, M, C' \rangle)\}$
            end if
          end if
        end for
        if $m = f(m_1, \ldots, m_\ell)$ **then**
          $C'' \leftarrow C \cup \{[n] \stackrel{\mathbf{c}}{\vdash} m_1, \ldots, [n] \stackrel{\mathbf{c}}{\vdash} m_\ell, \ldots, \}$
          $R \leftarrow R \cup \{\langle \hat{\mathscr{L}}, M, C'' \rangle\}$
        end if
        return R *; case f*
      else if $m \notin \mathcal{X}$ **then** *; When $[n]$ not necessarily a basis of itself*
        return EXPAND$(\hat{g}, n)$ *; case g*
      end if
    end if
  end for
end for
return $\{\hat{g}\}$ *; case h — Fixpoint*

Figure 5.9 Function REDUCE-STEP.

the constraint is trivially fulfilled for any solution to C. Therefore, the constraint can be removed from the set.

f) In this case, there exists a constraint $[n] \overset{\mathbf{c}}{\vdash} m$ in C such that for all $\sigma \in \mathbf{sol}(C)$, $\sigma(\textsc{quasi-basis}([n]))$ is a proper basis of $\sigma([n])$. Therefore, for all $\sigma \in \mathbf{sol}(C)$, $\sigma(m) \in \mathbf{NFTC}(\sigma(\textsc{quasi-basis}([n])))$. But if this is the case, then either $\sigma(m) \in \sigma(\textsc{quasi-basis}([n]))$, or $\sigma(m) = f(t_1, \ldots, t_k)$ and $t_1, \ldots, t_k \in \sigma(\textsc{quasi-basis}([n]))$. These two cases are handled by the code that spans until the return statement annoteted by 'case f'. In the first case, because $\sigma(m) \in \sigma(\textsc{quasi-basis}([n]))$, there must be elements of QUASI-BASIS$([n])$ that unify with m. For every unifying element, the constraint is removed and the unifier applied. The justification for this is similar than that in the case of the constraints $a \overset{\mathbf{c}}{=} b$. In the second case the constraint $[n] \overset{\mathbf{c}}{\vdash} m$ is split to k constraints concerning the submessages $t_1, \ldots, t_k$. Together these cover all the cases.

g) In this case, there exists a constraint $[n] \overset{\mathbf{c}}{\vdash} m$ in C but so that $\sigma(\textsc{quasi-basis}([n]))$ is not necessarily a basis of $\sigma([n])$. The correctness of the algorithm now depends on the correctness of EXPAND. That will be discussed below.

h) Finally, here $\hat{g}$ is returned as it stands. If the procedure reaches this point, then obviously the requirements for C to be in its simple normal form are fulfilled. Also, there must be at least one solution to C, because the only requirements left are inequality requirements and derivability requirements concerning single variables, that may be assigned arbitrarily chosen values.

Thus, REDUCE-STEP is an admissible symbolic state one-to-many transformation. Furthermore, if the input $\hat{g}$ is returned unmodifed, then $\hat{g}$ is in the simple normal form and must denote at least one nonsymbolic global state. Thus the requirements R1 and R2 are fulfilled.

Function EXPAND

The function EXPAND is presented in Figure 5.10. The purpose of EXPAND is the following: if a set $[n]$ with respect to the constraint set C in a symbolic global state $\hat{g}$ is such that there may be solutions σ for which $\sigma(\textsc{quasi-basis}([n]))$ is not a basis of $\sigma([n])$, then the simple method for solving derivability constraints (shown in REDUCE-STEP) does not work. In this case, the condition of $\sigma(\textsc{quasi-basis}([n]))$ being a basis of $\sigma([n])$ must be somehow enforced.

function EXPAND$(\hat{g}, n)$
  $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$
  $R \leftarrow \{\langle \hat{\mathscr{L}}, M, C \cup \{\overset{\mathbf{c}}{B}(n)\}\rangle\}$
  $\{b_1, \ldots, b_k\} \leftarrow$ QUASI-BASIS$([n])$
  for all $m \in \{b_1, \ldots, b_k\}$ **do**
    for all $d \in \mathscr{D}$ **do**
      if $d = P_{\text{symenc}}(e, d)$ and $m = e(x, y)$ **then**
        $C' \leftarrow \text{shift}(C, n) \cup \{[n] \overset{\mathbf{c}}{\vdash} x, [n+1] \overset{\mathbf{c}}{=} [n] \cup \{x\}, \overset{\mathbf{c}}{B}(n)\}$
        $R \leftarrow R \cup \{\langle \hat{\mathscr{L}}, M, C' \rangle\}$
      end if
      if $d = P_{\text{pubenc}}(e, d, p, s)$ and $m = e(x, y)$ **then**
        $C' \leftarrow \text{shift}(C, n) \cup \{[n] \overset{\mathbf{c}}{\vdash} x', x' \overset{\mathbf{c}}{=} s(x''), x \overset{\mathbf{c}}{=} p(x''), [n+1] \overset{\mathbf{c}}{=} [n] \cup \{x'\}, \overset{\mathbf{c}}{B}(n)\}$
        $R \leftarrow R \cup \{\langle \hat{\mathscr{L}}, M, C' \rangle\}$
      end if
    end for
  end for
  return R

Figure 5.10 Function EXPAND, a function called from REDUCE-STEP.

EXPAND works like a counterpart of QUASI-BASIS. While QUASI-BASIS may not make any extra assumptions about the substitutions that the set of messages will be subjected to, EXPAND explicitly makes some such assumptions in order to get a wider quasi-basis. For example, if $P_{\text{symenc}}(e, d) \in \mathscr{D}$ and $e(x, y)$ belongs to $[n]$, then EXPAND makes the assumption that x is derivable from $[n]$. If this assumption held the QUASI-BASIS algorithm would add y to the basis being built, because now $d(x, e(x, y))$ would belong to the algebraic closure. EXPAND performs this imaginary step of QUASI-BASIS.

The details involve the function shift. $\text{shift}(C, n)$ returns C but so that message set numbers larger than n are increased by one (and the corresponding derivability constraints updated). This 'makes room' for a new set definition with the number $n+1$. Then the requirement that a certain message is known is added with respect to $[n]$, and the element x known by assumption is added as $[n+1] \stackrel{\mathbf{c}}{=} [n] \cup \{x\}$. This complication is necessary. If we were just add x to the set $[n]$, the following could happen: Assume $e(x, x)$ belongs to $[n]$. EXPAND decides to require that x is known, and adds $[n] \stackrel{\mathbf{c}}{\vdash} x$ to C. But now, because x is known, the term $e(x, x)$ can be decrypted, and therefore $x \in [n]$. But after this, the derivability constraint is trivially fulfilled. This is a cyclic deduction that is not realizable; this is why the function shift is used.

The trick why EXPAND does not get into an infinite loop (after all, it could assume e.g. that a single variable x would be some compound term so that it could be perhaps then splitted) is that it can rely on the *chronological property of* REDUCE. This property is described in the next subsection.

Anyway, expand thus makes some explicit assumptions about messages known by the environment. The result is a set of symbolic global states that together represent the same set of global states as the original state.

The chronological property

In the context of the analysis process, the following property holds for the algorithm REDUCE: if $\hat{g}$ is any symbolic global state met during the execution of REDUCE and C is the corresponding constraint set, then if $[n]$ contains any variable $x \in \mathcal{X}$ and a constraint $[n] \stackrel{\mathbf{c}}{\vdash} t$ is being currently processed, then x is mentioned in a constraint $[k] \stackrel{\mathbf{c}}{\vdash} x$ for $k < n$.

This property holds because whenever the analysis process introduces a new variable x into an constraint set, it is either added in the form of a constraint $[n] \stackrel{\mathbf{c}}{\vdash} x$ for some n, or is added to the set by replacing some variable by a term containing possibly new variables. In the former case, the variable can appear only later in message sets, i.e. if $x \in [k]$ then $k > n$. This is due to the way the symbolic state space enumeration works — unfortunately a forward reference is necessary here. And in the latter case, the same argument holds, now inductively.

Then, because the derivability constraints are processed in their chronological order — this is where the name 'chronological property' comes — i.e. $[n] \overset{\mathbf{c}}{\vdash} t$ before $[n+k] \overset{\mathbf{c}}{\vdash} t$, $k > 0$, all derivability constraints earlier than any current one are already transformed to the single-variable form, which is the form where all the derivability constraints are in the simple normal form. (Recall that this is the form where a constraint set will be when REDUCE-STEP returns a fixpointed symbolic global state.)

(This property was mentioned in the discussion of EXPAND.)

5.5 Symbolic State Subsumption

Definition 5.16 Let $g = \langle \mathscr{L}, M \rangle \in \mathcal{G}$. The set of *monotonic extensions* of g, denoted by $\mathbf{mext}(g)$, is the set

$$\{g' \mid g' = \langle \mathscr{L}, M' \rangle \in \mathcal{G}, M \subseteq M', |M'| \in \mathbb{N}\}. \qquad \square \tag{5.20}$$

Definition 5.17 The symbolic state subsumption relation $\preccurlyeq$ over $\hat{\mathcal{G}} \times \hat{\mathcal{G}}$ is defined as

$$\hat{g}' \preccurlyeq \hat{g} \Leftrightarrow_{\mathrm{def}} (\forall g' \in \rho(\hat{g}'))[(\exists g \in \rho(\hat{g}))[g \in \mathbf{mext}(g')]]. \tag{5.21}$$

If $\hat{g}' \preccurlyeq \hat{g}$, then $\hat{g}'$ is said to be subsumed by $\hat{g}$. Similarly, $\hat{g}$ is said to subsume $\hat{g}'$. (The terminology goes here in the other direction — with respect to the direction of the relation symbol — than for substitutions, unfortunately.) $\square$

Knowing that $\hat{g}' \preccurlyeq \hat{g}$ may be important. If a symbolic state space search has encountered $\hat{g}$ previously and arrives then at $\hat{g}'$, it can stop immediately because all the states represented by $\hat{g}'$ — or their monotonic extensions — have been already visited in the form of $\hat{g}$. Examining a monotonic extension of a state instead of the state itself is always admissible because the analyzed system behaves monotonically with respect to the set of messages known by the environment.[1]

[1] In essence, security properties that hold when a larger set of known messages also hold when the set is smaller. This is because a larger set of known messages only gives the environment more possibilities to try to violate the security assertions. This is not actually

In this section, we develop a method that is able in some cases to prove that a symbolic global state is subsumed by another one. The method is not necessarily complete in the sense that it is not able to correctly decide $\hat{g}' \overset{?}{\preccurlyeq} \hat{g}$ in arbitrary cases. However, because the method will be used for the purpose mentioned, i.e. to reduce state space, it does not need to be. Completeness might be nice, but a complete procedure could be computationally too complex so that it would cost more to search the reduced state space than the one not reduced. Therefore, instead of returning 'yes' or 'no', our method here returns either 'yes', 'no' or 'not known'. 'Not known' means that it might be that one of the states subsumes another, but it might also be that this is not the case. In the case of 'not known', it must be assumed that the subsumption relation does not hold and thus a state space reduction is not applied.

The development here is restricted to symbolic global states such that the corresponding constraint sets are in their simple normal forms. It should be noted that this is indeed a restriction, because it might be that some *sets* of symbolic global states subsume others, whereas we are here concentrating solely on the subsumption relation between two *single* symbolic global states.

Lemma 5.16 *Let $\hat{g}, \hat{g}' \in \hat{\mathcal{G}}$. In order to prove that for $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$ and $\hat{g}' = \langle \hat{\mathscr{L}}', M', C' \rangle$, $\hat{g}' \preccurlyeq \hat{g}$, i.e. that $\hat{g}'$ is subsumed by $\hat{g}$, it is necessary and sufficient to show that*

1. *for all $n \in \mathcal{N}$, $\hat{\mathscr{L}}_1(n) = \hat{\mathscr{L}}'_1(n)$ (correspondence of processes); that*
2. *for all $n \in \mathcal{N}$, $\hat{\mathscr{L}}_3(n) = \hat{\mathscr{L}}'_3(n)$ (correspondence of nonce counters); that*
3. *for every realizing substitution σ' of $\hat{g}'$, there is a realizing substitution σ of $\hat{g}$ such that for all $n \in \mathcal{N}$ and $v \in \mathcal{V}$, $\sigma'(\hat{\mathscr{L}}'_2(n)(v)) = \sigma(\hat{\mathscr{L}}_2(n)(v))$ and that $\mathbf{AC}_{\mathscr{M}}(\sigma'(M')) \subseteq \mathbf{AC}_{\mathscr{M}}(\sigma(M))$.*

Proof. By definition. □

The question is now how to prove (1)–(3) in Lemma 5.16 when two symbolic global states are given. We employ the following method. In order to try to show that $\hat{g}' \preccurlyeq \hat{g}$, do the following. (Let $\hat{g}' = \langle \hat{\mathscr{L}}', M', C' \rangle$ and $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$.)

1. Check for the conditions 1.–2. in Lemma 5.16. If at least one of the conditions is violated, return 'no' and terminate immediately. Otherwise proceed to the next step. (This is a complete step.)

true in our model for terminal predicates of the form $\underline{\neg \mathbf{S}(t)}$ but such predicates are not interesting and are actually replaced with $\underline{\top}$. See Section 5.7.

2. Find a most general substitution $\sigma_1 : \mathcal{X} \mapsto \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ such that

$$(\forall n \in \mathcal{N}, v \in \mathcal{V})[\sigma_1(\mathscr{L}_2(n)(v)) = \mathscr{L}_2'(n)(v)]. \tag{5.22}$$

   In essence, functions $\mathscr{L}_2(n)$ are unified one-sidedly element-wise with the functions $\mathscr{L}_2'(n)$. If this cannot be done, return 'no'. Otherwise proceed to the next step. (This is a complete step. Namely, if σ_1 does not exist, then there exists at least one pair n, v such that the set $\{\sigma(\mathscr{L}_2(n)(v)) \mid \sigma \in \mathbf{real}(\hat{g})\}$ does not contain the set $\{\sigma(\mathscr{L}_2'(n)(v)) \mid \sigma \in \mathbf{real}(\hat{g}')\}$. Because C and C' can be assumed to be in their simple normal forms, every variable left in C and C' denote an infinite number of terms.) This step, when successful, will actually unify all the variables between C and C'. The reason is that all variables that remain in a constraint set that is in its simple normal form represent subterms of messages sent by the environment. But all such messages are permanently stored into the state variables of the participants as state variables cannot be assigned twice. (This argument depends however on the way how the symbolic state space enumeration works, and thereby can be verified only after the enumeration method has been explicitly specified.)

3. Applying σ_1 to C, it may be that $\sigma_1(C)$ is not in its simple normal form. Therefore, let $G = \textsc{reduce}(\sigma_1(\hat{g}))$. Perform the following steps then for all $\hat{g} \in G$. Return 'yes' if 'yes' is returned for at least one element of G. Otherwise return 'not known'.

4. Check that $\textsc{quasi-basis}(M') \setminus \mathcal{X} \subseteq \textsc{quasi-basis}(M) \setminus \mathcal{X}$ (this is the M of an $\hat{g} \in G$; therefore σ_1 has been applied to it already). If not, return 'no'. Otherwise proceed to the next step. (This is a complete step because $\textsc{quasi-basis}$ computes the smallest complete and full quasi-basis of the given set. Because C and C' can be assumed to be in their simple normal forms, all variables may assume in essence arbitrary values and thus there exists solutions that are essentially quasi-free substitutions. Moreover, because of the chronological property, single variables can be removed from the quasi-bases. Then, if $\textsc{quasi-basis}(M) \setminus \mathcal{X}$ does not contain $\textsc{quasi-basis}(M') \setminus \mathcal{X}$, the algebraic closure of $\sigma(M)$ for some such quasi-free substitution cannot contain that of $\sigma(M')$. However, observe that the split that took place in the previous step is a possibly incomplete step.)

5. Finally, try to show that that for every realizing substitution σ of $\hat{g}'$, σ is also a realizing substitution of $\hat{g}$. This is equivalent to checking that $\sigma \in \mathbf{sol}(C')$ implies $\sigma \in \mathbf{sol}(C)$. This can be done at least in some cases as follows.

   (a) That $a \overset{\mathbf{c}}{\not\approx} b \in C$ implies $a \overset{\mathbf{c}}{\not\approx} b \in C'$;

(b) and that $[n] \overset{\mathbf{c}}{\vdash} a \in C$ implies $[n] \overset{\mathbf{c}}{\vdash} a' \in C'$ such that $\textsc{quasi-basis}([n']) \setminus \mathcal{X} \subseteq \textsc{quasi-basis}([n]) \setminus \mathcal{X}$.

That a solution to C is a realizing substitution of $\sigma(\hat{g})$ follows then by noticing that the values of state variables are in their normal forms for all solutions of the corresponding constraint sets by assumption.

This completes the description of the method for proving the state subsumption relation in certain cases.

5.6 Enforcing Normal Forms

One of the assumptions made in the constraint solving process is that if $\sigma \in \mathbf{real}(\hat{g})$, then $\sigma(x)$ is in its normal form for all $x \in \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ appearing in $\hat{g}$.

This is a condition that needs to be explicitly enforced in certain cases that arise with the symbolic space enumeration method. However, in order to develop a transformation for doing this it is not necessary to know yet how the enumeration is done.

Thus, let us assume that $S \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ is a finite set of terms t for which it holds a priori that $\sigma(t)$ is in its normal form for for all $\sigma \in \mathbf{real}(\hat{g})$ for some $\hat{g}$. Let then $t \in \mathbf{T}_{\mathcal{S}}(S)$, i.e. be a term composed of constant terms and those in S. Obviously, $\sigma \in \hat{g}$ does not necessarily imply $\sigma(t) \in \mathscr{M}$ because t could have been chosen so that it does not appear in $\hat{g}$, and thus Definition 5.8 does not help.

We want now to introduce an admissible symbolic state one-to-many transformation that transforms a symbolic state $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$ and a term $t \in \mathbf{T}(S)$ into a set of symbolic states $\{\hat{g}_1, \ldots, \hat{g}_k\}$ and a set of terms $\{t_1, \ldots, t_k\} \subseteq \mathbf{T}_{\mathcal{S}}(\mathcal{X})$ such that (1) $\bigcup_{i \in \{1,\ldots,k\}} \rho(\hat{g}_i) = \rho(\hat{g})$; (2) $\sigma \in \mathbf{real}(\hat{g}_i)$ implies $\sigma(t_i) \in \mathscr{M}$; and (3) if the nonsymbolic global state corresponding to $\sigma \in \mathbf{real}(\hat{g})$ is g and $\sigma(t) = t_G$, then for all corresponding $\sigma_i \in \mathbf{real}(\hat{g}_i)$ such that σ_i corresponds to the same global state g, it holds that $\sigma_i(t) = \mathfrak{N}(t_G)$.

In essence, what the transformation does is the following. As $\hat{g}$ is a symbolic global states it represents a set of global states whose elements are obtained by substituting ground values for certain variables. Some of these substitutions can, when applied to the term t, result in a term that is not in its normal form, i.e. does not belong to $\mathscr{M}$. However, the function REDUCE and the accompanying machinery for handling symbolic global states assumes that this cannot happen. They do not, in particular, apply the rewrite system the instantiated terms. Now the transformation introduced herein actually performs all the necessary rewrite steps beforehand, so that when the transformation has been done, no more steps are necessary and thus the assumption of normal-formness holds.

Given any $t \in \mathbf{T}(S)$ we can think that t is obtained by composing it from smaller subterms, i.e. the subterms of t. Therefore it is enough to explicitly describe the required transformation for terms of form $t = f(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k \in (S \cup \mathbf{T}_{\mathcal{S}}(\emptyset))$, i.e. so that $t_i \in \mathscr{M}$ for all solutions of C. In other words, we proceed inductively.

Thus, let $t = f(t_1, \ldots, t_k)$ be given. Consider $\sigma(t)$ for $\sigma \in \mathbf{real}(\hat{g})$. Because $\sigma(t_1), \ldots, \sigma(t_k)$ are in their normal forms by assumption, if $\sigma(t) \notin \mathscr{M}$, then there must exists a top-level rewrite step $\sigma(t) \underset{\mathfrak{R}\ t}{\rightarrow} t'$. It is enough to take care of this single step.

The transformation required is computed by the function ENFORCE, which is presented in Figure 5.11. The notation is slightly ambiguous. There are two functions that are mutually recursive, namely ENFORCE and RECURSE. There is also a 'global' variable $\mathcal{S}$. It must be thought that for every invocation of ENFORCE, one instance of $\mathcal{S}$ is created. Therefore, the instances of RECURSE called from an instance of ENFORCE all update the same variable, but a new call to ENFORCE creates a new variable to update (think about the call to ENFORCE inside RECURSE as a 'black box' that does not affect the current computation, but returns the answer for a certain subterm in an oracle-like fashion).

ENFORCE calls ENFORCE-1, shown in Figure 5.12, which performs the step of guaranteeing that $\sigma(f(t_1, \ldots, t_k))$ will be in its normal form if it is already guaranteed that this holds for $t_1, \ldots, t_k$. This is actually the procedure where the real work is done; ENFORCE is simply a driver that applies ENFORCE-1 recursively.

Proving that ENFORCE-1 works correctly is easy but dulling, therefore a proof is omitted.

5.7 Symbolic Terminal Predicates

In the previous chapter, a syntax and semantics for terminal predicates was defined. This section concentrates on the implementation of the terminal predicates in the symbolic state space.

function RECURSE$(C, S, t_1, \ldots, t_k, i)$
if $i = k + 1$ **then**
  $\mathcal{S} \leftarrow \mathcal{S} \cup$ ENFORCE-1$(C, f(t_1, \ldots, t_k))$
else
  $R \leftarrow$ ENFORCE(C, S, t_i)
  for all $\langle C, t \rangle \in R$ **do**
    RECURSE$(C, S, t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_k, i + 1)$
  end for
end if
function ENFORCE(C, S, t)
if $t \in (S \cup \mathbf{T}_{\mathcal{S}}(\emptyset))$ **then**
  return $\langle C, t \rangle$
else
  $t = f(t_1, \ldots, t_k)$ *; Because* $\mathcal{X} \subseteq S$
  $\mathcal{S} \leftarrow \emptyset$
  RECURSE$(C, S, t_1, \ldots, t_k, 1)$
  return $\mathcal{S}$
end if

Figure 5.11 Function ENFORCE.

5.7.1 Calculating the Value of $\hat{\tau}$

In the previous chapter, a syntax and semantics for terminal predicates was defined. As mentioned, for a given terminal predicate τ, we need a symbolic version of it, $\hat{\tau}$, such that

$$\hat{\tau}(\hat{g}) \Leftrightarrow \forall g \in \rho(\hat{g}) : \tau(g) \tag{5.23}$$

holds.

We will now develop a function Q that, given a terminal symbolic global state $\hat{g}$, computes a set of symbolic global states $Q(\hat{g})$ such that

$$\bigcup_{\hat{g}' \in Q(\hat{g})} \rho(\hat{g}') \neq \emptyset \Leftrightarrow \neg\hat{\tau}(\hat{g}). \tag{5.24}$$

Let $\tau \in \Phi_\tau$ be a terminal predicate. Let $N(\tau)$ denote the quantifier-free form of τ where all negations are moved immediately before the atomic propositions using De Morgan's laws and where subformulae of the form $\neg\mathbf{S}(m)$ are replaced with $\top$ (true). In essence, $N(\tau)$ is τ rewritten using the following rewrite rules:

1. $\neg(\tau_1 \wedge \tau_2) \rightarrow \neg\tau_1 \vee \neg\tau_2$

function ENFORCE-1(C, t)
$\mathcal{S} \leftarrow \emptyset$
$t = f(t_1, \ldots, t_k)$
for all $P_{n\text{-tuple}}(c, s_1, \ldots, s_n) \in D$ where $f = s_i$ and $k = 1$ **do**
  for $i = 1$ to n **do**
    $v_i \leftarrow \text{NEWVAR}(C)$
  end for
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C \cup \{t_1 \stackrel{\mathbf{c}}{=} c(v_1, \ldots, v_n)\}, v_i\rangle\}$
  $C \leftarrow C \cup \{t_1 \stackrel{\mathbf{c}}{\not\approx} c(*_1, \ldots, *_n)\}$
end for
for all $P_{\text{inv}}(f, g) \in D$ where $k = 1$ **do**
  $v \leftarrow \text{NEWVAR}(C)$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C \cup \{t_1 \stackrel{\mathbf{c}}{=} g(v)\}, v\rangle\}$
  $C \leftarrow C \cup \{t_1 \stackrel{\mathbf{c}}{\not\approx} g(*_1)\}$
end for
for all $P_{\text{symenc}}(e, f) \in D$ where $k = 2$ **do**
  $v \leftarrow \text{NEWVAR}(C)$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C \cup \{t_2 \stackrel{\mathbf{c}}{=} e(t_1, v)\}, v\rangle\}$
  $C \leftarrow C \cup \{t_2 \stackrel{\mathbf{c}}{\not\approx} e(t_1, *_1)\}$
end for
for all $P_{\text{pubenc}}(e, f, p, s) \in D$ where $k = 2$ **do**
  $v_1 \leftarrow \text{NEWVAR}(C)$
  $v_2 \leftarrow \text{NEWVAR}(C)$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C \cup \{t_1 \stackrel{\mathbf{c}}{=} s(v_1), t_2 \stackrel{\mathbf{c}}{=} e(p(v_1), v_2)\}, v_2\rangle\}$
  $C \leftarrow C \cup \{t \stackrel{\mathbf{c}}{\not\approx} f(s(*_1), e(p(*_1), *_2))\}$
end for
for all $P_{\text{signs}}(c, f, p, s) \in D$ where $k = 3$ **do**
  $v \leftarrow \text{NEWVAR}(C)$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C \cup \{t_2 \stackrel{\mathbf{c}}{=} p(v), t_3 \stackrel{\mathbf{c}}{=} c(s(v), t_1)\}, 1\rangle\}$
  $C \leftarrow C \cup \{t \stackrel{\mathbf{c}}{\not\approx} f(*_1, p(*_2), c(s(*_2), *_1))\}$
end for
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle C, t\rangle\}$
return $\mathcal{S}$

Figure 5.12 Function ENFORCE-1.

2. $\underline{\neg(\tau_1 \vee \tau_2)} \to \underline{\neg\tau_1 \wedge \neg\tau_2}$

3. $\underline{\neg\neg\tau} \to \underline{\tau}$

4. $\underline{\neg\exists r\tau} \to \underline{\forall r\tau}$

5. $\underline{\neg\forall r\tau} \to \underline{\exists r\tau}$

6. $\underline{\forall r\tau} \to \underline{\tau[n_1/r] \wedge \cdots \wedge \tau[n_k/r]}$, where $\mathscr{N} = \{n_1, \ldots, n_k\}$. Here $\tau[x/y]$ denotes τ with all free occurrences of y replaced with x.

7. $\underline{\exists r\tau} \to \underline{\tau[n_1/r] \vee \cdots \vee \tau[n_k/r]}$

8. $\underline{\forall r\tau} \to \underline{\tau[n_1/r] \wedge \cdots \wedge \tau[n_k/r]}$

9. $\underline{\neg\mathbf{S}(m)} \to \underline{\top}$

The motivation for removing subformulae of the form $\underline{\neg\mathbf{S}(m)}$ is that there is no idea in checking for such a property in a security protocol, and therefore (1) it would be waste of time to implement a method for checking for such cases, and (2) most probably the occurrence of such a requirement in the normal form of a terminal predicate corresponds to an error in the terminal predicate's original specification.

The idea is now to construct a function that, given a terminal predicate τ and a symbolic global state $\hat{g} = \langle\hat{\mathscr{L}}, M, C\rangle$, computes a collection of constraint sets such that in order for $\neg\hat{\tau}(\hat{g})$ to be true, at least one of the sets in the collection must have a solution. For example, if $\tau = \underline{\mathbf{S}(m)}$, then the corresponding constraint set would be $\{[n] \overset{\mathbf{c}}{\vdash} \hat{\hat{\Upsilon}}(\hat{g}, m)\}$ ($\hat{\hat{\Upsilon}}$ is defined below in complete analogy with Definition 4.15), assuming that $[n] = M$ and that $\hat{\hat{\Upsilon}}(\hat{g}, m)$ is in its normal form for all the realizing substitutions of $\hat{g}$. Namely, because $\neg\hat{\tau}$ must be true, there must be a state in $\rho(\hat{g})$ for which $\underline{\mathbf{S}(m)}$ is false. But this predicate is false exactly if m is known by the intruder, i.e. if $M \vdash m$. This correspond to the additional constraint presented.

What must be done now is to **(1)** define how the propositions of terminal predicates map to constraints, and **(2)** how these constraint sets are combined for complex terminal predicates.

Definition 5.18 Analogously to Definition 4.15, we define the symbolic version of $\tilde{\Upsilon}$, namely $\hat{\tilde{\Upsilon}} : \hat{\mathcal{G}} \times \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}}) \mapsto \mathbf{T}_{\mathcal{S}}(\mathcal{X})$:

$$\hat{\tilde{\Upsilon}}(\hat{g}, t) = \begin{cases} t, & \text{if } t \in \mathbf{T}_{\mathcal{S}}(\emptyset) \\ \mathscr{L}_2(n)(v), & \text{if } t = n : v \\ f(\hat{\tilde{\Upsilon}}(\hat{g}, t_1), \dots, \hat{\tilde{\Upsilon}}(\hat{g}, t_k)), & \text{otherwise when } t = f(t_1, \dots, t_k).\square \end{cases} \tag{5.25}$$

Definition 5.19 Let $t \in \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}})$. We define $V(t)$ to be the set of elements of $\mathcal{V}_{\mathcal{N}}$ that appear in t. $\square$

Definition 5.20 Let $t \in \mathbf{T}_{\mathcal{S}}(\mathcal{V}_{\mathcal{N}})$. We define $E(t)$ to be the set $\textsc{enforce}(\emptyset, V(t), t)$. $\square$

Definition 5.21 Let A and B be sets of sets. We define

$$A \oplus B =_{\text{def}} \{a \cup b \mid a \in A, b \in B.\} \qquad \square \tag{5.26}$$

Definition 5.22 Let $\hat{g} = \langle \hat{\mathscr{L}}, M, C \rangle$ be a global state and τ a terminal predicate consisting of a single proposition possibly negated. We define

$$P(\hat{g}, \tau) = \begin{cases} \{\{C \cup C' \cup \{t \overset{\mathbf{c}}{\not\approx} t'\}\} \\ \quad \mid \langle C, t \rangle \in E(\hat{\tilde{\Upsilon}}(\hat{g}, m)), \langle C', t' \rangle \in E(\hat{\tilde{\Upsilon}}(\hat{g}, m'))\} \\ \qquad\qquad\qquad\qquad \text{if } \tau = \underline{m = m'} \\ \{\{C \cup C' \cup \{t \overset{\mathbf{c}}{=} t'\}\} \\ \quad \mid \langle C, t \rangle \in E(\hat{\tilde{\Upsilon}}(\hat{g}, m)), \langle C', t' \rangle \in E(\hat{\tilde{\Upsilon}}(\hat{g}, m'))\} \\ \qquad\qquad\qquad\qquad \text{if } \tau = \underline{\neg(m = m')} \\ \{\{C \cup C' \cup \{[k] \overset{\mathbf{c}}{\vdash} t\}\} \mid \langle C, t \rangle \in E(\hat{\tilde{\Upsilon}}(\hat{g}, m))\} \\ \qquad\qquad\qquad\qquad \text{if } \tau = \underline{\mathbf{S}(m)} \text{ and } M = [k].\square \end{cases} \tag{5.27}$$

Definition 5.23 Let τ be an arbitrary terminal predicate in its quantifier-free normal form, i.e. $\tau = N(\tau)$. We define

$$M(\hat{g}, \tau) = \begin{cases} P(\hat{g}, \tau) & \text{if } \tau \text{ is a single proposition} \\ P(\hat{g}, \tau) & \text{if } \tau = \underline{\neg \tau'} \text{ and } \tau' \text{ is a proposition} \\ M(\hat{g}, \tau_1) \cup M(\hat{g}, \tau_2) & \text{if } \tau = \underline{\tau_1 \vee \tau_2} \\ M(\hat{g}, \tau_1) \oplus M(\hat{g}, \tau_2) & \text{if } \tau = \underline{\tau_1 \wedge \tau_2}.\ \square \end{cases} \tag{5.28}$$

M is a function from normal-form terminal predicates and symbolic global states to collections of constraint sets. Defining now

$$Q(\hat{g} = \langle \mathscr{L}, M, C \rangle) = \{\langle \mathscr{L}, M, C \cup C' \rangle \mid C' \in M(\hat{g}, \tau), \} \tag{5.29}$$

(5.24) is fulfilled for terminal symbolic global states.

5.7.2 Terminal Predicate Rewriting

It is possible that certain subpredicates of a terminal predicate became identically true or false during state space enumeration before a terminal state has been reached. For example, if two participants have assigned the same term to certain corresponding state variables, and a subpredicate of a given terminal predicate requires these values to be distinct, then it can be immediately recognized that the value of the subpredicate has become identically false.

It is important to notice that such subpredicates can be replaced by their truth values and then the terminal predicate simplified. In this vein, it is possible (and probable) that on some branches of the search, the terminal predicate becomes identically true or false before the actual terminal states are reached. Thus, the value of $\hat{\tau}$ can be determined without explicitly visiting the terminal states. This can bring large benefits with respect to analysis time.

Another, smaller benefit from simplifying terminal predicates as early as possible is that the predicates get smaller and thus working with them later in the search becomes cheaper.

Chapter 6

Symbolic Search and State Space Reductions

6.1 Symbolic Successor Function

The actual search method is described. Covered topics are the symbolic successor function, traces, steps and their symbolic counterparts, a symmetry method and the actual algorithms that implement the state space enumeration.

In order to be able to actually do symbolic state space enumeration, a symbolic successor function is needed. That is, the function s mentioned in (4.27).

We want to now develop a symbolic successor function for our particular symbolic representation. We first define an auxiliary function.

Definition 6.1 We define the function $\mathscr{A} : \hat{\mathcal{G}} \times \mathcal{N} \mapsto 2^{\hat{\mathcal{G}}}$ as follows. Let $\hat{g} \in \hat{\mathcal{G}} = \langle \hat{\mathscr{L}}, M, C \rangle$ and $n \in \mathcal{N}$. Let $a = \hat{\mathscr{L}}(n) = \langle p, \hat{\Upsilon}, c \rangle$. The value of $\mathscr{A}(\hat{g}, n)$ is defined in cases depending on p.

1. If $p = \mathbf{halt}_n$ then $\mathscr{A}(\hat{g}, n) = \emptyset$.
2. If $p = \urcorner v.p'$ then

$$\mathscr{A}(\hat{g}, n) = \{\langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}, c \rangle], M \cup \{\hat{\Upsilon}(v)\}, C \cup \{[k+1] \stackrel{\mathbf{c}}{=} [k] \cup \{\hat{\Upsilon}(v)\}\}\rangle\} \tag{6.1}$$

for k the largest integer for which $[k] \neq \perp$ in C.

3. If $p = \Rsh v.p'$ then

$$\mathscr{A}(\hat{g}, n) = \{\langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}[v \mapsto x], c\rangle], M, C \cup \{[k] \overset{\mathbf{c}}{\vdash} x\}\rangle\} \tag{6.2}$$

where k is the largest integer for which $[k] \neq \perp$ in C (this $[k]$ corresponds to M) and x is a new variable $x \in \mathcal{X}$ so that x does not appear in $\hat{g}$.

4. if $p = \uparrow v.p'$ then

$$\mathscr{A}(\hat{g}, n) = \{\langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}[v \mapsto n.c], c + 1\rangle], M, C\rangle\}. \tag{6.3}$$

5. If $p = v \leftarrow t.p'$ then let S be the total image of $\hat{\Upsilon}$, i.e.

$$S = \bigcup_{u \in \mathcal{V} : \hat{\Upsilon}(u) \neq \perp} \hat{\Upsilon}(u). \tag{6.4}$$

Let $E = \textsc{enforce}(C, S, \hat{\Upsilon}(t))$. Then

$$\mathscr{A}(\hat{g}, n) = \{\langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}[v \mapsto x], c\rangle], M, C'\rangle \mid \langle C', x\rangle \in E\}. \tag{6.5}$$

6. If $p = (v = v')?p' : p''$ then

$$\begin{aligned} \mathscr{A}(\hat{g}, n) = \; & \{\langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}, c\rangle], M, C \cup \{\hat{\Upsilon}(v) \overset{\mathbf{c}}{\doteq} \hat{\Upsilon}(v')\}\rangle, \\ & \langle \hat{\mathscr{L}}[n \mapsto \langle p'', \hat{\Upsilon}, c\rangle], M, C \cup \{\hat{\Upsilon}(v) \overset{\mathbf{c}}{\not\approx} \hat{\Upsilon}(v')\}\rangle\}. \end{aligned} \tag{6.6}$$

There are no other cases. □

Lemma 6.1 *Let* $\hat{g} = \langle \hat{\mathscr{L}}, \hat{M}, C\rangle \in \hat{\mathcal{G}}$ *be a symbolic global state so that for the largest* k *so that* $[k] \neq \perp$ *in* C, $\mathbf{AC}_{\mathscr{M}}(\sigma([k])) = \mathbf{AC}_{\mathscr{M}}(\sigma(M))$ *for all* $\sigma \in$ $\mathbf{real}(\hat{g})$.[1] *Let* $n \in \mathcal{N}$. *Let* $\hat{\mathscr{L}}(n) = \langle p, \hat{\Upsilon}, c\rangle$. *Let* $G = \rho(\hat{g})$ *be the set of the global states represented by* $\hat{g}$. *For all* $\langle \mathscr{L}, M\rangle \in G$, $\mathscr{L}_1(n) = p$ *for some fixed* p. *Let* $G' = \{g' \mid g \in G \wedge g \underset{n.p}{\longrightarrow} g'\}$. *Let then* $\hat{G} = \mathscr{A}(\hat{g}, n)$. *Then*

$$\bigcup_{\hat{g}' \in \hat{G}} \rho(\hat{g}') = G'. \tag{6.7}$$

Proof. That p is fixed is immediate from our definition of a symbolic global state. The remaining proof is given in cases depending on p.

We denote $\bigcup_{\hat{g}' \in \hat{G}} \rho(\hat{g}')$ by R, wanting thus prove that $R = G'$.

[1]Intuitively, $[k]$ represents M. The analysis process creates symbolic states for which this holds. See Definition 6.1.

1. $p = \mathbf{halt}_n$. Then $\mathscr{A}(\hat{g}, n) = \emptyset$. Thus $R = \emptyset$. From Definition 4.12 it is seen that for $g \in G$, $\nexists g' : g \underset{n.p}{\longrightarrow} g'$. Therefore also $G' = \emptyset$. Thus $R = G'$.

2. $p = \Lsh v.p'$. Then there is (by Definition 4.12) a one-to-one correspondence $\xi : G \mapsto G'$ such that

$$\xi(\langle \mathscr{L}, M \rangle) = \langle \mathscr{L}[n \mapsto \langle p', \Upsilon, c \rangle], M \cup \{\Upsilon(v)\} \rangle \tag{6.8}$$

   when $\mathscr{L}(n) = \langle p, \Upsilon, c \rangle$.

   We know that $\hat{G} = \{\hat{g}'\}$ where

$$\hat{g}' = \langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}, c \rangle], \hat{M} \cup \{\hat{\Upsilon}(v)\}, C \cup \{[k+1] \overset{\mathbf{c}}{=} [k] \cup \{\hat{\Upsilon}(v)\}\} \rangle. \tag{6.9}$$

   by Definition 6.1.

   Obviously, $\mathbf{real}(\hat{g}) = \mathbf{real}(\hat{g}')$ because no new terms have been added to $\hat{g}'$ and the extra constraint added does not restrict the possible solutions to C. Let then $\sigma \in \mathbf{real}(\hat{g})$ be arbitrarily chosen and $\rho_\sigma(\hat{g})$ the particular unique nonsymbolic state corresponding to this realizing substitution. Let similarly $\rho_\sigma(\hat{g}')$ be the particular unique nonsymbolic state corresponding to the same realizing substitution when applied to $\hat{g}'$. Then if

$$\rho_\sigma(\hat{g}) = \langle \mathscr{L}, M \rangle \tag{6.10}$$

   for which $\mathscr{L}(n) = \langle p, \Upsilon, c \rangle$, we have $\sigma(\hat{\mathscr{L}}) = \mathscr{L}$ and $\sigma(\hat{M}) = M$. Then $\sigma(\hat{M} \cup \{\hat{\Upsilon}(v)\}) = M \cup \Upsilon(v)$ and $\sigma(\hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}, c \rangle]) = \mathscr{L}[n \mapsto \langle p', \Upsilon, c \rangle]$. Therefore $\rho_\sigma(\hat{g}') = \xi(\rho_\sigma(\hat{g}))$. Thus, the same one-to-one correspondence ξ that holds between G and G' holds also for the elements of G and R. Therefore $R = G'$.

3. $p = \Rsh v.p'$. We know that $\hat{G} = \{\hat{g}'\}$ where

$$\hat{g}' = \langle \hat{\mathscr{L}}[n \mapsto \langle p', \hat{\Upsilon}[v \mapsto x], c \rangle], \hat{M}, C \cup \{[k] \overset{\mathbf{c}}{\vdash} x\} \rangle \tag{6.11}$$

   where k is the largest integer for which $[k] \neq \perp$ in C and x is a new variable $x \in \mathcal{X}$ so that x does not appear in $\hat{g}$, by Definition 6.1.

   On the other hand, we know that G' contains exactly all those states g' that are of the form

$$\langle \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto m], c \rangle], M \rangle \tag{6.12}$$

   for some $\langle \mathscr{L}, M \rangle \in G$ and for which $M \vdash m$.

We see that every element σ' of $\mathbf{real}(\hat{g}')$ is $[x \mapsto m]\circ\sigma$ for some $m \in \sigma(M)$ and $\sigma \in \mathbf{real}(\hat{g})$. On the other hand, every such σ' belongs to $\mathbf{real}(\hat{g}')$.

We show first $R \subseteq G'$. Assume $g' \in R$. Then $g' = \sigma'(\hat{g}')$ for some $\sigma' \in \mathbf{real}(\hat{g}')$. We can let $\sigma' = [x \mapsto m] \circ \sigma$ by the argument above so that $\sigma(\hat{g}) \in G$. This means that if $g' = \langle \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto m], c\rangle], M\rangle$ then $g = \langle \mathscr{L}, M\rangle$ and belongs to G. Then $g' \in G'$, because $M \vdash m$.

We then show $G' \subseteq R$. Let $g' = \langle \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto m]c,]\rangle, M\rangle$ belong to g'. Then $g = \langle \mathscr{L}, M\rangle$ belongs to G. Therefore there exists $\sigma \in \mathbf{real}(\hat{g})$ such that $\sigma(\langle \hat{\mathscr{L}}, \hat{M}\rangle) = g$. This solution can be extended to $\sigma' = [x \mapsto m] \circ \sigma$. Then this is, by the argumentation above and the assumption that $\mathbf{AC}_{\mathscr{M}}(\sigma([k])) = \mathbf{AC}_{\mathscr{M}}(\sigma(M))$, an element of $\mathbf{real}(\hat{g}')$. Applying this substitution to $\hat{g}'$ then yields g'. Therefore $G' \subseteq R$.

Thus $G' = R$.

4. $p =\uparrow v.p'$. This case is the easiest one and is left as an exercise to the reader.

5. $p = v \leftarrow t.p'$. From Definition 6.1 we see that $\hat{G}$ is now a set consisting an arbitrary number of elements. However, by the definitions of ENFORCE and of a noncontradictory constraints set, R corresponds to the set

$$\rho(\langle \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto \Upsilon(t)], c\rangle], M, C\rangle). \tag{6.13}$$

Because here no extra constraints are added the result $G' = R$ is easy to see; the details are analogous to the previous cases.

However, one might ask why the function ENFORCE is then involved at all. The reason is not in the semantics of the symbolic states but in the reduction process that is used when we use $\mathscr{A}$ to really implement the symbolic state space enumeration method.

6. $p = (v = v')?p' : p''$. We know, by Definition 4.12, that there exists then a one-to-one correspondence $\xi : G \mapsto G'$ such that

$$\xi(\langle \mathscr{L}, M\rangle) = \begin{cases} \langle \mathscr{L}[n \mapsto \langle p', \Upsilon, c\rangle], M\rangle & \text{if } \Upsilon(v) = \Upsilon(v'), \text{ and} \\ \langle \mathscr{L}[n \mapsto \langle p'', \Upsilon, c\rangle], M\rangle & \text{otherwise, when } \mathscr{L}(n) = \langle p, \Upsilon, c\rangle. \end{cases} \tag{6.14}$$

Also, by Definition 6.1, $\hat{G} = \{\hat{g}_1, \hat{g}_2\}$ so that

$$\hat{g}_1 = \langle \mathscr{L}[n \mapsto \langle p', \Upsilon, c\rangle], M, C \cup \{\Upsilon(v) \stackrel{\mathbf{c}}{=} \Upsilon(v')\}\rangle \tag{6.15}$$

and

$$\hat{g}_2 = \langle \mathscr{L}[n \mapsto \langle p'', \Upsilon, c\rangle], M, C \cup \{\Upsilon(v) \overset{\mathbf{c}}{\not\approx} \Upsilon(v')\}\rangle. \tag{6.16}$$

We see that every realizing substitution $\sigma \in \mathbf{real}(\hat{g})$ is either a realizing substitution of $\hat{g}_1$ or of $\hat{g}_2$ but never of the both. This is because $\sigma(x) = \sigma(y)$ and $\sigma(x) \neq \sigma(y)$ cannot hold at the same time, and on the other hand neither $\Upsilon(v)$ nor $\Upsilon(v')$ can contain elements of $\mathscr{K}$ (the pattern variables).

Therefore it is easy to see that ξ is also a one-to-one correspondence between the states in G and R. Therefore $G' = R$.

Thus, in all cases, $G = R'$. The proof is complete. □

Definition 6.2 The symbolic successor function $\mathscr{S} : \hat{\mathcal{G}} \mapsto 2^{\hat{\mathcal{G}}}$ is defined as follows:

$$\mathscr{S}(\hat{g}) = \bigcup_{n \in \mathcal{N}} \mathscr{A}(\hat{g}, n), \tag{6.17}$$

where $\mathscr{A}$ is the function defined in Definition 6.1. □

Lemma 6.2 *Let $\hat{g} \in \hat{\mathcal{G}}$. Let $\rho(\hat{g}) = G$. Let $G' = \{g' \mid \exists g \in G : g \to g'\}$. Then*

$$\bigcup_{\hat{g}' \in \mathscr{S}(\hat{g})} \rho(\hat{g}') = G'. \tag{6.18}$$

Proof. This result follows easily from Definition 6.2 and Lemma 6.1. □

Definition 6.3 We define the labeled symbolic transition relation $\xrightarrow[\ell]{s}$ over $\hat{\mathcal{G}} \times \hat{\mathcal{G}}$ as

$$\langle \mathscr{L}, M, C\rangle \xrightarrow[n.p]{s} \hat{g}' \Leftrightarrow (\mathscr{L}_1(n) = p \wedge \hat{g}' \in \mathscr{A}(\hat{g}, n)). \tag{6.19}$$

The union

$$\bigcup_{\ell} \xrightarrow[\ell]{s} \tag{6.20}$$

is denoted by $\xrightarrow{s}$. □

Lemma 6.3 *Let $\hat{g} \in \hat{\mathcal{G}}$. Let $\rho(\hat{g}) = G$. Then*

$$\bigcup_{\hat{g}' : \hat{g} \xrightarrow{s} \hat{g}'} \rho(\hat{g}') = \{g' \mid \exists g \in G : g \to g'\}. \tag{6.21}$$

Proof. This follows now from Definition 6.3 and the Lemma 6.2. □

Thus, we have now a well-defined symbolic transition relation that implements the semantics of the nonsymbolic transition relation in the symbolic world.

6.2 Quasi-Symmetries

With the constraint solver and the symbolic representation, an infinite problem has been transformed to a finite one. This categorical reduction is necessary, but after it the state space explosion looms. Our verification problem is **NP**-hard (see Appendix A) and thus we can expect every algorithm for solving it to behave exponentially in some cases. Still, methods for reducing the state space as much as possible can be useful. In our work, some state space reductions are based on the general notion of a quasi-symmetry.

Let $(S, \to)$ be a state space, i.e. a directed graph, and $\sigma : S \mapsto 2^S$ be the corresponding successor function such that if $s \in S$, then

$$s \to s' \Leftrightarrow s' \in \sigma(s). \tag{6.22}$$

Let $T \subseteq S$ be a set of states. Consider now the following problem: given a state $s \in S$, does there exist $t \in T$ such that $s \to^* t$? A simple way to solve this is to start from s and enumerate all states reachable from s. However, there can be states in the set of states enumerated that are so 'close' to anothers that enumerating all of them 'seems' to be irrelevant.

Let us assume that we have a transitive and reflexive binary relation $\preccurlyeq$ over S such that the following holds:

$$s \preccurlyeq s' \Rightarrow ((\exists t \in T)[s \to^* t] \Rightarrow (\exists t' \in T)[s' \to^* t']). \tag{6.23}$$

Thus, if a state in T is reachable from s then a state in T is also reachable from s'. We call any such a relation a T-*quasi-symmetry.*

Consider now the algorithm QSSEARCH in Figure 6.1, which is a naïve way of extending a standard state space enumeration algorithm to employ a quasi-symmetry for reducing the effective state space size. The idea of QSSEARCH is

function QSSEARCH($s_0, \sigma, \preccurlyeq$)

```
V ← ∅
W ← {s_0}
while W ≠ ∅ do
    Choose s ∈ W
    W ← W \ {s}
    V ← V ∪ {s}
    if s ∈ T then
        T-state reached. Stop.
    end if
    for all s' ∈ σ(s) do
        if ¬(∃s'' ∈ (V ∪ W))[s' ≼ s''] then
            W ← W ∪ {s'}
        end if
    end for
end while
No T-state reachable from s_0. Stop.
```

Figure 6.1 Reduced state space enumeration using a quasi-symmetry.

that whenever we encounter a state s' such that s'' has already been encountered for which $s' \preccurlyeq s''$, then we can stop search at s' because if s' will lead to a T-state then will s'' also. Thus it is enough to continue search from s''.

Calling the algorithm naïve is, unfortunately, not without reason. If $(S, \rightarrow)$ is a generic state space and $\preccurlyeq$ can be whatever relation fulfilling (6.23), then the algorithm may fail to produce the correct answer. The reason is that the algorithm can prune the search space too eagerly and leave nothing in the active search set.[2] However, for our purposes the following lemma is enough.

Lemma 6.4 *Let $(S, \rightarrow)$ be a finite state space whose graph is a directed tree with the successor function σ. Let $s_0 \in S$ and let $d(s)$ denote the length of the path from s_0 to s when it exists. Let $T \subseteq S$ be given and $\preccurlyeq$ be a quasi-symmetry over S (w.r.t T) such that*

$$s \preccurlyeq s' \Rightarrow d(s) \leq d(s'). \tag{6.24}$$

Then QSSEARCH($s_0, \sigma, \preccurlyeq$) *produces always the correct answer to the question whether or not a T-state is reachable from s_0.*

Proof. Let a *reduction* be a pair $\langle s', s'' \rangle$ such that s' was not added to the

[2] The reader might find it amusing to try to find a simple example exhibiting this behavior. It is easy.

working set W on the 12th line of the algorithm because s'' was in $V \cup W$ and $s' \preccurlyeq s''$. Let us now assume that QSSEARCH has terminated. Clearly, if no T-state is reachable from s_0, then also QSSEARCH returns 'false' and works thus correctly. Assume then that at least one T-state is reachable from s_0 and that the algorithm has terminated. Let R be the set of reductions made. Let Q be the (finite) set of maximal linear paths enumerated, i.e. the maximal set of maximal paths starting from s_0 and visiting only states that have once been in W or whose immediate predecessors have been in W. Every such path starts from s_0 and ends either in a terminal state or a state that was reduced to another state, i.e. to a state s such that $\langle s, s' \rangle \in R$. Let $\sqsubset$ be a binary relation over Q such that $q_1 \sqsubset q_2$ iff q_1 ends to a state s such that $\langle s, s' \rangle \in R$ and R appears in q_2.

The reflexive and transitive closure $\sqsubseteq$ of $\sqsubset$ is a partial order (we show anti-symmetry): For the sake of contradiction, assume that $q_1 \sqsubset q_2 \sqsubset \cdots \sqsubset q_k = q_1$ and let the corresponding reduced states be $s_1, \ldots, s_k$. In essence, for all $i \in \{1, \ldots, k-1\}$ it holds that

$$(\exists s_i^* \in q_{i+1})[\langle s_i, s_i^* \rangle \in R]. \tag{6.25}$$

But for any such pair s, s^*, necessarily $d(s) \leq d(s^*)$ because $s \preccurlyeq s^*$. On the other hand, $d(s_i^*) < d(s_{i+1})$ because s_i^* has been in W and thus is a proper predecessor of s_{i+1}. Thus we get $d(s_1) < d(s_2) < \cdots < d(s_1)$ which is a contradiction. Therefore $\sqsubseteq$ is antisymmetric.

Therefore $\sqsubseteq$ is a partial order and any chain $q_1 \sqsubset q_2 \sqsubset \cdots$ is finite. Let Q_S be the set of all maximal paths starting from s_0 and leading to a terminal state. We show that at least one $q \in Q_S$ such that q ends in a T-state is fully contained in Q. Namely, let q be a chain in Q_S such that q ends in a T-state. Either $q \in Q$ or then a prefix of q is in Q. In the first case the lemma obviously holds. Consider the second case. Because q leads to a T-state, at least one of the paths q' such that $q \sqsubset q'$ either leads to a T-state or is a prefix of path leading to a T-state. The result follows now by induction over the length of the chain $q \sqsubset q_1 \sqsubset \cdots$ where if $q_k \sqsubset q_{k+1}$ appears in the chain then q_{k+1} is a (proper or trivial) prefix of a path leading to a T-state. Because this chain is necessarily finite, induction succeeds. $\square$

6.3 Traces and steps

6.3.1 Nonsymbolic Traces and Steps

Definition 6.4 Let $g_1, \ldots, g_k \in \mathcal{G}$ be nonsymbolic global states such that

$$g_1 \underset{\ell_1}{\longrightarrow} g_2 \underset{\ell_2}{\longrightarrow} \cdots \underset{\ell_{k-1}}{\longrightarrow} g_k. \tag{6.26}$$

We say that the states $g_1, \ldots, g_k$ together constitute a *trace* from g_1 to g_k that has the *trace label* $\langle \ell_1, \ldots, \ell_{k-1} \rangle$. The length of the trace is $k-1$, i.e. the number of transitions. In particular, from every $g \in \mathcal{G}$ there is a unique trace of zero length with the trace label $\langle\rangle$. □

Definition 6.5 A trace $T = \langle g_1, \ldots, g_k \rangle$ with the trace label $L = \langle n_1.p_1, \ldots, n_{k-1}.p_{k-1} \rangle$ is a *step* iff

(1) $n_i = n_j$ for all $i, j \in \{1, \ldots, k-1\}$, and

(2) for all $i \in \{1, \ldots, k-1\}$, if $p =\!\!\Rsh\; v.p'$, $g_i = \langle \mathscr{L}, M, N \rangle$ and $g_1 = \langle \mathscr{L}', M', N' \rangle$, then $\mathbf{AC}_{\mathscr{M}}(M') = \mathbf{AC}_{\mathscr{M}}(M)$.

A *maximal step* T is a step starting from g such that T is not a proper prefix of any step starting from g.

If there is a maximal step T from g to g' with the trace label L, we denote

$$g \underset{L}{\rightsquigarrow} g'. \tag{6.27}$$

The union $\bigcup_L \underset{L}{\rightsquigarrow}$ is denoted by $\rightsquigarrow$. This is the *maximal step transition relation.*

Lemma 6.5 *Let $g, g' \in \mathcal{G}$ such that g' is a terminal state. Then $g \rightarrow^* g'$ iff $g \rightsquigarrow^* g'$.*

Proof. The proof is by mathematical induction on the length k of traces from g to g'. For $k = 0$ the lemma holds trivially. Now assume that the lemma holds for traces of length k. We show that it holds also for traces of length $k+1$. Namely, let T be a trace of length $k+1$ from g_1 to g_{k+2} and L the corresponding trace label. By the induction hypothesis, we may assume that the trace from g_2 to g_{k+2} corresponds to a sequence of maximal steps, starting from g_2.

Let $L = \langle n_1.p_1, n_2.p_2, \ldots, n_{k+1}.p_{k+1} \rangle$.

If n_1 does not appear in the set $\{n_2, \ldots, n_k\}$, then the trace label $\langle n_1.p_1\rangle$ corresponds to a maximal step from g_1 and we are done.

Otherwise, let use denote for every i: $g_i = \langle \mathscr{L}_i, M_i\rangle$. If $\mathbf{AC}_{\mathscr{M}}(M_2) = \mathbf{AC}_{\mathscr{M}}(M_1)$, then the action corresponding to the label $n_1.p_1$ can be postponed to take place immediately before the next action by the same principal n_1, because whatever the action does, it cannot be detected by the other principals, and, if it is receive, it can be executed also later because the knowledge of the environment increases monotonically. In essence, let k be the smallest index such that $n_k = n_1$. We reorder actions so that

$$g_1 \xrightarrow[n_1.p_1]{} g_2 \rightarrow^* g_k \xrightarrow[n_k.p_k]{} g_{k+1} \cdots \tag{6.28}$$

becomes

$$g_1 \xrightarrow[n_2.p_2]{} g_2' \rightarrow^* g_{k-1}' \xrightarrow[n_1.p_1]{} g_k \cdots \tag{6.29}$$

The resulting trace corresponds to a sequence of maximal steps.

The remaining case is the case where $\mathbf{AC}_{\mathscr{M}}(M_2) \supset \mathbf{AC}_{\mathscr{M}}(M_1)$. Now we can move all actions by n_1 up to, but not including, the first receive performed by n_1 (if such a receive exists) to follow immediately after the first one. This can be done because in the moved actions there are no receives, and thus making them execute earlier can only cause monotonic increase in the sets M_i. After this, the resulting trace corresponds again to a sequence of maximal steps. This concludes the induction step and the proof is complete. □

6.3.2 Symbolic Traces and Steps

The concepts of a trace and a step can be incorporated into the symbolic space also.

Definition 6.6 Let $\hat{g}_1, \ldots, \hat{g}_k \in \hat{\mathcal{G}}$ be symbolic global states such that

$$\hat{g}_1 \overset{s}{\underset{\ell_1}{\rightarrow}} \hat{g}_2 \overset{s}{\underset{\ell_2}{\rightarrow}} \cdots \overset{s}{\underset{\ell_{k-1}}{\rightarrow}} \hat{g}_k. \tag{6.30}$$

We say that the states $\hat{g}_1, \ldots, \hat{g}_k$ together constitute a *symbolic trace* from $\hat{g}_1$ to $\hat{g}_k$ that has the trace label $\langle \ell_1, \ldots, \ell_{k-1}\rangle$. The length of the trace is $k-1$, i.e. the number of symbolic transitions. In particular, from every $g \in \hat{\mathcal{G}}$ there is a unique symbolic trace of zero length with the trace label $\langle\rangle$. □

Definition 6.7 Let $\hat{T} = \langle \hat{g}_1, \ldots, \hat{g}_k\rangle$ be a symbolic trace with the trace label $L = \langle n_1.p_1, \ldots, n_{k-1}.p_{k-1}\rangle$. Then $\hat{T}$ is a symbolic step if

1. $n_i = n_j$ for all $i, j \in \{1, \dots, k-1\}$, and
2. For all $i \in \{1, \dots, k-1\}$, if $p_i =\!\!\Rsh\ v.p'$, $\hat{g}_i = \langle \mathscr{L}, M, C\rangle$ and $\hat{g}_1 = \langle \mathscr{L}', M', C'\rangle$, then QUASI-BASIS$(M) =$ QUASI-BASIS(M').

A symbolic step is a *maximal symbolic step* if it is not a proper prefix of any symbolic step.

If there is a maximal symbolic step from $\hat{g}$ to $\hat{g}'$ with the trace label L we denote

$$\hat{g} \underset{L}{\overset{s}{\leadsto}} \hat{g}'. \tag{6.31}$$

The union $\bigcup_L \underset{L}{\overset{s}{\leadsto}}$ is denoted by $\overset{s}{\leadsto}$. This is the *maximal symbolic step transition relation.* □

Lemma 6.6 *Let $\hat{g}, \hat{g}' \in \hat{\mathcal{G}}$ so that $\hat{g}'$ is a terminal symbolic state and so that $\hat{g} \overset{s}{\rightarrow}^* \hat{g}'$. Let $g \in \hat{g}'$ be arbitrarily chosen. Then there exists $\hat{g}'' \in \hat{\mathcal{G}}$ so that $g \in \rho(\hat{g}'')$ and $\hat{g} \overset{s}{\leadsto}^* \hat{g}''$.*

Proof. Choose $g_1 \in \hat{g}$ so that $g_1 \rightarrow^* g$. Then there exists, by Lemma 6.5, a trace

$$g_1 \rightarrow g_2 \rightarrow \cdots \leadsto g_k = g \tag{6.32}$$

that consists of maximal steps alone. Now there must exist a symbolic trace

$$\hat{g} = \hat{g}_1 \overset{s}{\rightarrow} \hat{g}_2 \overset{s}{\rightarrow} \cdots \overset{s}{\rightarrow} \hat{g}_k \tag{6.33}$$

so that $g_i \in \rho(\hat{g}_i)$ for $i \in \{1, \dots, k\}$. Let $T_1, \dots, T_n$ be the set of maximal steps in the trace $\langle g_1, \dots, g_k\rangle$ so that $\langle g_1, \dots, g_k\rangle = T_1 \S \cdots \S T_k$, where § denotes concatenation of the traces. (More precisely, $\langle g_1, \dots, g_k\rangle \S \langle g_k, \dots, g_\ell\rangle = \langle g_1, \dots, g_\ell$. Observe that g_k appears in both of the traces.) We now show that if $T_i = \langle g_a, \dots, g_b\rangle$, then $\hat{g}_a \overset{s}{\leadsto}^* \hat{g}_b$. Then naturally $\hat{g} \overset{s}{\leadsto}^* \hat{g}_k$. The results then follows by setting $\hat{g}'' = \hat{g}_k$.

It is enough to show that if $T_i = \langle g_a, \dots, g_b\rangle$ and $T_{i+1} = \langle g_b, \dots, g_c\rangle$ then there cannot be a maximal symbolic step transition $g_m \overset{s}{\leadsto} g_n$ such that $m < b$ and $n > b$.

We show this. Because T_i and T_{i+1} correspond to maximal steps, it must hold that $g_b \underset{n.p}{\longrightarrow} g_{b+1}$ where $p =\!\!\Rsh\ v.p'$ for some v and p', $g = \langle \mathscr{L}, M\rangle$, and $\mathbf{AC}_{\mathscr{M}}(M) \neq \mathbf{AC}_{\mathscr{M}}(M')$ where $g_a = \langle \mathscr{L}', M'\rangle$.

Therefore there must be an integer $a \leq x < b$ such that $g_x \to g_{x+1}$, $g_x = \langle \mathscr{L}, M \rangle$, $g_{x+1} = \langle \mathscr{L}', M' \rangle$ and $\mathbf{AC}_{\mathscr{M}}(M) \subset \mathbf{AC}_{\mathscr{M}}(M')$, and so that for the trace $g_{x+1} \xrightarrow[\ell_{x+1}]{} \cdots \xrightarrow[\ell_{b-1}]{} g_b$ none of the labels $\ell_{x+1}, \ldots, \ell_{b-1}$ is of the form $n. \Rsh v.p$ for some p.

Assume then that there exists the maximal symbolic step transition $g_m \rightsquigarrow g_n$. Then $m \leq x$ because the transition is maximal. But because there exist $\langle \mathscr{L}, M \rangle \in \rho(\hat{g}_x)$ and $\langle \mathscr{L}, M' \rangle \in \rho(\hat{g}_{x+1})$ so that $\mathbf{AC}_{\mathscr{M}}(M) \subset \mathbf{AC}_{\mathscr{M}}(M')$, then if $\hat{g}_x = \langle \hat{\mathscr{L}}, \hat{M}, C \rangle$ and $\hat{g}_{x+1} = \langle \hat{\mathscr{L}}', \hat{M}', C' \rangle$, it must hold that $\textsc{quasi-basis}(\hat{M}) \neq \textsc{quasi-basis}(\hat{M}')$ (otherwise this could not be possible). Then the definition of symbolic steps (Definition 6.7) leads to $m \leq b$ which is a contradiction. □

6.4 Symmetries

Definition 6.8 Let τ be a terminal predicate over $\mathcal{G}$ and let R be a binary relation over $\mathcal{G} \times \mathcal{G}$ such that

$$g \; R \; g' \Leftrightarrow ((\exists g_1 \in \mathcal{G}_T)[g \to^* g_1 \wedge \tau(g_1)] \Leftrightarrow (\exists g_2 \in \mathcal{G}_T)[g' \to^* g_2 \wedge \tau(g_2)]). \tag{6.34}$$

Then R is said to be a *τ-symmetry*. R is easily seen to be an equivalence relation. □

Lemma 6.7 *Let $\Pi_{\mathcal{N}}$ be the set of all permutations of $\mathcal{N}$, i.e. the set of all bijective functions $\mathcal{N} \mapsto \mathcal{N}$. Let τ be a fixed terminal predicate. Then the relation $\underset{\mathbf{sym}}{\approx}$ over $\mathcal{G} \times \mathcal{G}$, defined as*

$$g \underset{\mathbf{sym}}{\approx} g' \Leftrightarrow_{\mathrm{def}} (\exists \pi \in \Pi_{\mathcal{N}})[\pi(g) = g' \wedge \pi(\tau) = \tau] \tag{6.35}$$

is a τ-symmetry.

$\pi(g)$ denotes here the replacement of all occurrences of $n \in \mathcal{N}$ in g with $\pi(n)$. Similarly for $\pi(\tau)$. More exactly, let $g = \langle \mathscr{L}, M \rangle$ and $g' = \langle \mathscr{L}', M' \rangle$. Let $\sigma : \mathbf{T}_{\mathcal{S}}(\emptyset) \mapsto \mathbf{T}_{\mathcal{S}}(\emptyset)$ be defined as

$$\sigma(x) = \begin{cases} \pi(n).c & \textit{if } x = n.c \\ f(\sigma(t_1), \ldots, \sigma(t_k)) & \textit{if } x = f(t_1, \ldots, t_k) \\ x & \textit{otherwise.} \end{cases} \tag{6.36}$$

Then

$$M' = \{\sigma(m) \mid m \in M\} \tag{6.37}$$

and $\mathscr{L}'$ is defined so that

$$\mathscr{L}'(n) = \langle p, \sigma(\Upsilon), c\rangle \tag{6.38}$$

whenever

$$\mathscr{L}(\pi^{-1}(n)) = \langle p, \Upsilon, c\rangle \tag{6.39}$$

(here $\sigma(\Upsilon)$ is σ's obvious extension from $\mathbf{T}_{\mathcal{S}}(\emptyset) \mapsto \mathbf{T}_{\mathcal{S}}(\emptyset)$ to $(\mathcal{V} \mapsto \mathbf{T}_{\mathcal{S}}(\emptyset)) \mapsto (\mathcal{V} \mapsto \mathbf{T}_{\mathcal{S}}(\emptyset))$).

Proof. Applying $\pi \in \Pi_{\mathcal{N}}$ to g corresponds to simply renaming the participants in the protocol with new names. This renaming is transparent because the set $\mathcal{N}$ is not ordered in any way and the renaming is made so that the terminal predicate is also symmetric with respect to it. Therefore the states g and $\pi(g)$ cannot be distinguished between by the analysis process, and therefore, if $g' = \pi(g)$, the states g and g' cannot. This implies the claim. □

6.4.1 Symbolic Symmetries

We define the relation $\underset{\mathbf{sym}}{\approx}$ also over symbolic states, i.e. over $\hat{\mathcal{G}} \times \hat{\mathcal{G}}$. The definition is

$$\begin{aligned}\hat{g} \underset{\mathbf{sym}}{\approx} \hat{g}' \Leftrightarrow_{\text{def}} \quad & (\forall g \in \rho(\hat{g}))[(\exists g' \in \rho(\hat{g}'))[g \underset{\mathbf{sym}}{\approx} g']]\wedge \\ & (\forall g \in \rho(\hat{g}'))[(\exists g' \in \rho(\hat{g}))[g \underset{\mathbf{sym}}{\approx} g']].\end{aligned} \tag{6.40}$$

Obviously, some method is needed for computing this relation. One way to do this is the following: guess an element π of $\Pi_{\mathcal{N}}$. Show first that $\pi(\tau) = \tau$ for the given terminal predicate, and then that

$$\pi(\hat{g}) \preccurlyeq_c \pi(\hat{g}') \tag{6.41}$$

and that

$$\pi(\hat{g}') \preccurlyeq_c \pi(\hat{g}). \tag{6.42}$$

Showing both would imply $\hat{g} \underset{\mathbf{sym}}{\approx} \hat{g}'$ in the sense of the definition above.

However, it is somehow wasteful to use the $\preccurlyeq_c$ relation in this way. Indeed, it would be better do define a generalized subsumption relation *that takes symmetries into account*. We define this relation, subsumption with symmetries, as follows:

$$\hat{g} \underset{\mathbf{sym}}{\preccurlyeq} \hat{g}' \Leftrightarrow_{\mathrm{def}} (\exists \pi \in \Pi_{\mathcal{N}})[\pi(\hat{g}) \preccurlyeq_c \hat{g}' \wedge \pi(\tau) = \tau]. \tag{6.43}$$

This relation is a τ-quasi-symmetry for all terminal predicates τ in the sense of (6.23). It can be proven in some cases using the method described in Section 5.5 combined with guessing the symmetry permutation. Naturally, the guess has better to be an educated one.

6.4.2 Symbolic Symmetries and Terminal Predicate Rewriting

As mentioned previously, it can be useful to rewrite the terminal predicate on-the-fly as the search progresses. If this is done, then different branches of the state space actually have different terminal predicates.

In this case, the definition of the symmetry relation must be adapted to take the different terminal predicates into account. Then, instead of (6.43), we must use symmetry defined actually over $\left(\hat{\mathcal{G}} \times \Phi_\tau\right)^2$ as follows:

$$\langle \hat{g}, \tau \rangle \underset{\mathbf{sym}}{\preccurlyeq} \langle \hat{g}', \tau' \rangle \Leftrightarrow_{\mathrm{def}} (\exists \pi \in \Pi_{\mathcal{N}})[\pi(\hat{g}) \preccurlyeq_c \hat{g}' \wedge \pi(\tau) = \tau']. \tag{6.44}$$

We use this form in the search driver, which is the subject of the next section.

6.5 Search Driver

Now we are ready to develop our main search driver, the culmination of the work presented in the last few chapters. Basically we use the symbolic state space search outlined in Chapter 4. However, we extend the algorithm so that:

(i) Individual actions are not executed individually. Instead, symbolic steps are used.

(ii) The relation $\underset{\mathbf{sym}}{\preccurlyeq}$ is used to reduce state space.

(iii) Terminal predicates are simplified and rewritten on the fly in order recognize uninteresting search branches as early as possible.

function DRIVER($\hat{g}, \tau$)
$W \leftarrow \{\langle \hat{g}, \tau \rangle\}$
$V \leftarrow \emptyset$
while $W \neq \emptyset$ **do**
  Choose $\langle \hat{g}, \tau \rangle \in W$ that is on the smallest possible depth
  $W \leftarrow W \setminus \{\hat{g}, \tau\}$
  $R \leftarrow$ STEP-STATE($\hat{g}, \tau$)
  for all $\hat{g}' \in R$ **do**
    if $(\exists \langle \hat{g}'', \tau' \rangle \in (W \cup V))[\langle \hat{g}', \tau \rangle \underset{\mathbf{sym}}{\preccurlyeq} \langle \hat{g}'', \tau' \rangle]$ **then**
      $R \leftarrow R \setminus \{\hat{g}'\}$
    else if $(\exists \langle \hat{g}'', \tau' \rangle \in W)[\langle \hat{g}'', \tau' \rangle \underset{\mathbf{sym}}{\preccurlyeq} \langle \hat{g}', \tau \rangle]$ **then**
      $W \leftarrow W \setminus \{\hat{g}'', \tau'\}$
    end if
  end for
  for all $\hat{g}' \in R$ **do**
    $g' = \langle g', \tau' \rangle$
    $\tau' \leftarrow$ REWRITE($\hat{g}', \tau$)
    if $\tau' = \underline{\neg\top}$ **then**
      Report security violation and stop.
    else if $\tau' \neq \underline{\top}$ **then**
      $W \leftarrow W \cup \{\langle \hat{g}', \text{REWRITE}(\hat{g}', \tau') \rangle\}$
    end if
  end for
  $V \leftarrow V \cup \{\hat{g}, \tau\}$
end while

Figure 6.2 Function DRIVER, the main search driver.

Because using $\underset{\mathbf{sym}}{\preccurlyeq}$ requires us to remember some visited states, a depth-first search alone is not the best option. Observe that if $\hat{g} \underset{\mathbf{sym}}{\preccurlyeq} \hat{g}'$ then $\hat{g}$ and $\hat{g}'$ must be on the same depth in the state space. Therefore a breadth-first search seems natural. The width of the search tree will not probably be an issue.

The main search driver is shown in Figure 6.2. The driver, named DRIVER, performs breadth-first state space enumeration using the maximal symbolic step transition relation. The set of next symbolic states for a given state is computed by the function STEP-STATE. This function is shown in figure 6.3. REWRITE is a function that rewrites the given terminal predicate for the given symbolic global state, simplifying it if possible. If the simplification leads to an identically true or false terminal predicate, either the symbolic state is discarded or a security violation is reported, respectively. Basically, REWRITE

function STEP-STATE($\hat{g}, \tau$)
if $\hat{g}$ is terminal **then**
    Check the value of $\hat{\tau}(\hat{g})$, where $\hat{\tau}$ corresponds to τ.
    if $\neg\hat{\tau}(\hat{g})$ **then**
        Report security violation and stop
    else
        return $\emptyset$
    end if
else
    $R \leftarrow \emptyset$
    for all $n \in \mathcal{N}$ **do**
        $R \leftarrow R \cup$ STEP($\hat{g}, n, \hat{g}$)
    end for
end if
return R

Figure 6.3 Function STEP-STATE, called from DRIVER.

removes those propositions from a given terminal predicate whose values can be already determined, replacing them with their truth values. Then the terminal predicate is simplified using the standard simplification rules for propositional calculus. We do not give an explicit formulation.

The function STEP-STATE first checks if the given state if a terminal one. If it is, the value of $\hat{\tau}$ corresponding to the terminal predicate associated with the state is evaluated. Otherwise for every principal $n \in \mathcal{N}$, the corresponding maximal symbolic steps are calculated by the function STEP, shown in the Figure 6.4.

STEP is mutually recursive with XSTEP. The latter just for calls REDUCE for a given state and then gives the reduced states to STEP. XSTEP is shown in Figure 6.5.

STEP takes three arguments: the current state, the name of the principal and the symbolic state where the step started from. (The last argument is needed in order to check when the maximal step ends. See Definition 6.7.) Then STEP just performs recursive iteration of the function $\mathscr{A}$ until maximal step boundaries are found.

There is an important detail in the function STEP. On line 12, instead of merely checking whether QUASI-BASIS(M_0) = QUASI-BASIS(M), we check instead if QUASI-BASIS(M_0) $\setminus \mathcal{X}$ = QUASI-BASIS(M) $\setminus \mathcal{X}$. As an equivalence relation the latter is weaker, and thus is true more often. We can drop single variables from the quasi-bases away because when steps are computed, all constraint sets are

function STEP$(\hat{g}, n, \hat{g}_0)$
$\hat{g} = \langle \mathscr{L}, M, C \rangle$
$\mathscr{L}(n) = \langle p, \Upsilon, c \rangle$
if $p = \mathbf{halt}_k$ **then**
  return $\emptyset$
else if $p = \Lsh v.p'$ **then**
  $\mathscr{L}' \leftarrow \mathscr{L}[n \mapsto \langle p', \Upsilon, c \rangle]$
  $M' \leftarrow M \cup \Upsilon(v)$
  $C' \leftarrow C \cup \{[k+1] \stackrel{\mathbf{c}}{=} [k] \cup \{\Upsilon(v)\}\}$ for largest k such that $[k] \stackrel{\mathbf{c}}{=} [k-1] \cup S \in C$
  return STEP$(\langle \mathscr{L}', M', C \rangle, n, \hat{g}_0)$
else if $p = \Rsh v.p'$ **then**
  $\hat{g}_0 = \langle \mathscr{L}_0, M_0, C_0 \rangle$
  if QUASI-BASIS$(M_0) \setminus \mathcal{X} \neq$ QUASIBASIS$(M) \setminus \mathcal{X}$ **then**
    return $\hat{g}$
  else
    $x \leftarrow$ a new variable not appearing in $\hat{g}$
    $\mathscr{L}' \leftarrow \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto x], c \rangle]$
    $C' \leftarrow C \cup \{[k] \stackrel{\mathbf{c}}{\vdash} x\}$ for largest k such that $[k] \stackrel{\mathbf{c}}{=} [k-1] \cup S \in C$
    return XSTEP$(\langle \mathscr{L}', M', C' \rangle, n, \hat{g}_0)$
  end if
else if $p = \uparrow v.p'$ **then**
  $\mathscr{L}' \leftarrow \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto n.c], c+1 \rangle]$
  return STEP$(\langle \mathscr{L}', M, C \rangle, n, \hat{g}_0)$
else if $p = v \leftarrow t.p'$ **then**
  $S \leftarrow \bigcup_{u \in \mathcal{V} : \Upsilon(u) \neq \perp} \Upsilon(u)$
  $E \leftarrow$ ENFORCE$(C, S, \Upsilon(t))$
  $T \leftarrow \emptyset$
  for all $\langle C', x \rangle \in E$ **do**
    $T \leftarrow T \cup$ XSTEP$(\langle \mathscr{L}[n \mapsto \langle p', \Upsilon[v \mapsto x], c \rangle], M, C' \rangle, n, \hat{g}_0)$
  end for
  return T
else if $p = (v = v')?p : p'$ **then**
  $S_1 \leftarrow \langle \mathscr{L}[n \mapsto \langle p', \Upsilon, c \rangle], M, C \cup \{\Upsilon(v) \stackrel{\mathbf{c}}{=} \Upsilon(v')\} \rangle$
  $S_2 \leftarrow \langle \mathscr{L}[n \mapsto \langle p'', \Upsilon, c \rangle], M, C \cup \{\Upsilon(v) \stackrel{\mathbf{c}}{\not\approx} \Upsilon(v')\} \rangle$
  return XSTEP$(S_1, n, \hat{g}_0) \cup$ XSTEP$(S_2, n, \hat{g}_1)$
end if

Figure 6.4 Function STEP, called from STEP-STATE.

function XSTEP($\hat{g}, n, \hat{g}_0$)
1: $T \leftarrow$ REDUCE($\hat{g}$)
2: **for all** $\hat{g}' \in T$ **do**
3: $R \leftarrow R \cup$ STEP($\hat{g}', n, \hat{g}_0$)
4: **end for**
5: **return** R

Figure 6.5 Function XSTEP, mutually recursive with STEP.

in their simple normal forms. This is because REDUCE is called whenever a constraint set could be not in this form. Therefore every single variable in the message set of a symbolic global state actually represents a message derivable from the rest of the set because of the chronological property (see 5.4.1). Thus the single variables do not actually contribute to the algebraic closures of the message sets when realized.

This completes the description of the search driver. Putting all results from this and the previous chapters together, we find that DRIVER is a sound and complete symbolic state space search algorithm for our problem.

Part II

Implementation

Chapter 7

Implementation Overview

An overview of the implementation of given, including the language and tools used, and an illustration showing the graphical interface in use.

7.1 Language

We implemented the analysis method described in the first part as a tool for cryptographic protocol analysis. The tool is written in Scheme [37]. Scheme is a semi-functional programming language that is a well-established dialect of LISP. While [37] can be seen as a definitive description of the language, an IEEE standard also exists [35].

Scheme is typically an interpreted language. We used Matthias Blume's VSCM system. However, in order to get a stand-alone version of the tool we also wrote a compiler that compiles the subset of Scheme used in the tool to C. This compiler is not, however, part of the thesis work and is thus not described further here. Nevertheless the tool can be now run as a stand-alone program without any need for a Scheme interpreter.

7.2 Code Overview

The code base is around 6,000 lines of Scheme, plus around 1,000 lines from the SILex system which was not included in the previous figure. (SILex is "Scheme Implementation of Lex'; it is a lexical parser generator.)

Of the 6,000 lines, the parser uses 900 lines. Around 750 lines are used for miscellaneous functionality. The actual analysis process is thus implemented in ca. 4,300 lines of Scheme.

7.3 Used Algorithms

The algorithms used in the implementation are not so optimized as one could hope. Indeed, based on the experiments with the tool, it seems that analysis time could be significantly reduced by optimizing the core engine. We feel that especially the constraint solver does some redundant work.

However, even now the tool is reasonably fast. The analysis of the Needham-Schröder protocol takes around 20 seconds on a moderately fast Pentium box. Therefore we did not see optimization as a high-priority task for now.

7.4 Graphical Display

We also implemented an interactive program for displaying the generated state space during the analysis process. The display program can be connected to the tool. Then states appear in the display program as soon as they are generated by the tool. In this way, the analysis process can be illustrated.

The user of the program is able to retrieve essential data about the states from the display tool. Paths leading to insecure states are highlighted. A screen shot of the display tool is shown in Figure 7.1.

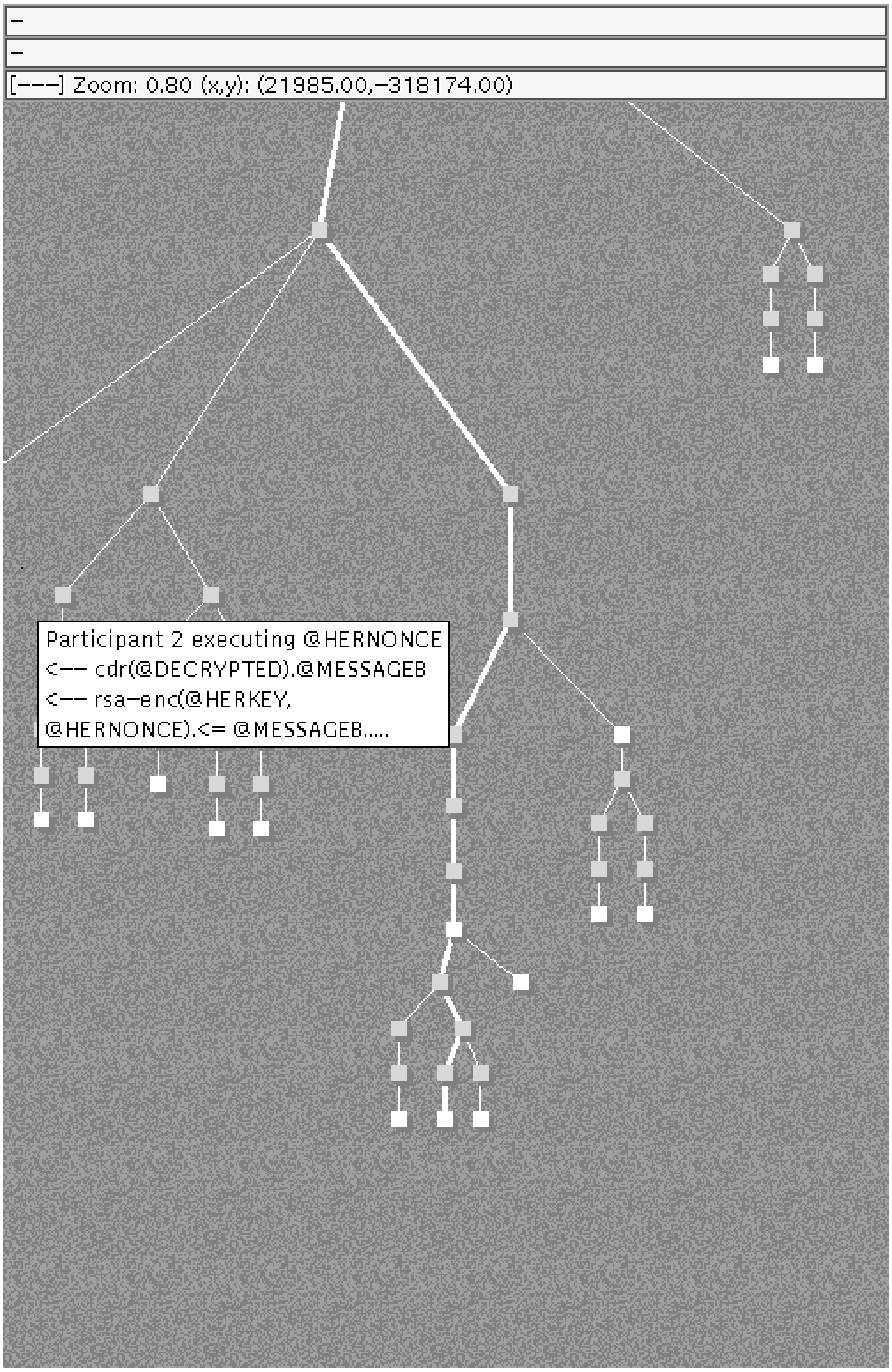

Figure 7.1 A screen shot of the display tool.

Chapter 8

The Input Language

The input language used in the tool is described. A BNF grammar of the language is given. The main aspects of transforming input programs to the formal model are presented with the aids of denotational semantics.

The analysis tool contains a parser that compiles a higher-level language into the formal model that has been described earlier. The language reflects the underlying model quite directly. The most important addition is the ability to define procedures and functions. They can be used to make protocol descriptions easier to understand and more concise.

The tool is used by giving it a file as input that contains a protocol description written in the language. The language contains also some syntax for declaring extra options. For example certain state space reductions can be enabled or disabled. These options are not a part of the protocol description and are thus not compiled into the model. Instead, they affect the analysis process directly in other ways.

In the next section we exhibit the features of the language. We then give a formal grammar for the language and proceed to describe how the procedures are mapped to the formal model.

8.1 Informal Exhibition

An input file consists of *pragmas*, *cryptosystem and data declarations*, *procedure definitions* and *principal instantiations*. These four parts must follow each other in this order.

8.1.1 Pragmas

Pragmas are declarations that control the way the tool works. They can be used to enable or disable generic options and to control the amount of debugging and other output from the tool. Generic options are set with

`pragma enable` *option*;

and unset with

`pragma disable` *option*;.

For example, to disable terminal predicate rewriting one can give

`pragma disable filter-early;`.

The output is controlled by the pragmas

`pragma enable-output` *path*;

and

`pragma disable-output` *path*;

where *path* denotes the specific class of messages whose output should be enabled or suppressed. *Path* is a list of colon-separated names. For example, to enable all the debugging messages from the 'search' module one can give

`pragma enable-output debug:search;`.

In a similar vein,

`pragma disable-output debug:search:expand;`

disables some more specific debugging output from the 'search' module. The paths are hierarchic as one can see and the most specific declaration holds for any output message.

8.1.2 Cryptosystem And Data Declarations

After the pragmas there come the cryptosystem and data declarations. All constants that are used in the protocol description must be declared in this section. First, atomic terms are declared with

`declare-atoms` *atomic*, *atomic*, ... ;

where *atomic* is a name for an atomic term. In the language, all constants are written with lowercase letters (and all variables with uppercase letters). Thus, one can e.g. give

```
declare-atoms a, b, c;
```

to declare that there exist at least the three atoms `a`, `b` and `c`. Multiple atom declarations can be given, i.e. multiple occurrences of `declare-atoms` are allowed. The same holds for all other declarations also.

Operators are declared with

`declare-operators` *operator*, *operator*, ... ;

in an analogous manner to `declare-atoms`. `declare-operators` is just used to declare the operators that can be used. Their properties, i.e. what rewrite rules they obey, are given later.

Those terms that are public, i.e. known by the environment at the very beginning of the protocol, are given by using

`declare-public` *term*, *term*, ... ;.

Implicitly, `true` belongs to this set in any case.

The properties of the operators are declared by the following constructs.

`declare-tuple` a `with-selectors` b_1, b_2, ... , b_k;

corresponds to $P_{k\text{-tuple}}(a, b_1, \ldots, b_k)$;

`declare-inverse` a `to` b;

corresponds to $P_{\text{inv}}(a, b)$;

declare-encryption a decrypted-by b;

corresponds to $P_{\text{symenc}}(a, b)$;

declare-PK-encryption a decrypted-by b
with-public-key-deriver c with-private-key-deriver d;

corresponds to $P_{\text{pubenc}}(a, b, c, d)$; and

declare-signature-system a verified-by b
with-signing-key-deriver c with-verification-key-deriver d;

corresponds to $P_{\text{signs}}(a, b, c, d)$.

For example, one could define the properties of the RSA cryptosystem by giving:

```
declare-operators RSAenc, RSApub, RSApriv, RSAkeym;
declare-PK-encryption                           RSAenc
                      decrypted-by              RSAenc
                      with-public-key-deriver   RSApub
                      with-private-key-deriver  RSApriv;
declare-PK-encryption                           RSAenc
                      decrypted-by              RSAenc
                      with-public-key-deriver   RSApriv
                      with-private-key-deriver  RSApub;
declare-inverse                                 RSAkeym
                      to                        RSApriv;
```

This contains the information that RSA encryption and decryption are basically the same operation; that from the private key one can compute the key material and thus the public key, and that the rôles of encryption and decryption can be exchanged.

8.1.3 Procedures

A protocol is described by defining one or more *procedures* and then creating principals that execute some of those procedures. Procedures may call other procedures, *but only those that precede them in the input file.* This restriction in particular removes any possibility of (mutually) recursive procedures.

A procedure is defined as

$$\texttt{procedure}\ \mathit{name}\ \texttt{[}arg_1\texttt{,}\ \ldots\texttt{]}\ \texttt{local}\ v_1\texttt{,}\ \ldots\ \{\ \mathit{statements}\ \}.$$

Arg_i's are a list of procedure arguments. They are variable names. V_i's are names of local, temporary variables. If the list of local variables is empty the keyword `local` can be dispensed with.

The procedure body, *statements*, is a sequence of actions to do. The available actions correspond quite directly with the constructs of the formal programs.

First,

$$\texttt{send}\ v\texttt{;}$$

denotes that the message contained in the variable v should be sent. Messages can be received by

$$\texttt{receive}\ v\texttt{;}.$$

This construct binds v to a message received from the environment. Similarly,

$$\texttt{generate}\ v\texttt{;}$$

denotes that a new nonce is generated and stored to v.

Procedure calls are written as

$$\mathit{procedure\text{-}name}\ \texttt{[}\ \mathit{term}\texttt{,}\ \ldots\ \texttt{]}\ \texttt{->}\ \mathit{variable}\texttt{,}\ \ldots\texttt{;}.$$

Here the procedure *procedure-name* is called with the arguments inside the square brackets. The return values are stored to the variables listed after the arrow. If the procedure returns no values the arrow can be dispensed with. Observe that procedures indeed are capable of returning multiple values.

Assignments to variables are denoted by

$$\mathit{variable}\ \texttt{:=}\ \mathit{term}\texttt{;}.$$

For example,

```
A := des-enc(K,secret);
```

binds the variable `A` to the term formed by the operator `des-enc`, the contents of the variable `K` and the constant `secret`.

Conditionals are written as

`if` v_1 `=` v_2 $\{$ *statements* $\}$ `else` $\{$ *else-statements* $\}$.

The `else`-part is optional and can be dispensed with. Arbitrary terms are *not* allowed in the condition but only simple variables. This is of course no restriction.

Return from a procedure is denoted by

`return` *term*`,` $\ldots$`;`

where the list of terms is the list of values to return. The length of this list must match the list of return variables in the corresponding procedure calls.

Total halt, i.e. end of protocol is denoted by

`halt;`.

A security assertion, i.e. a terminal predicate, can be associated with a halting point. This is accomplished by using

`halt-check` *predicate*`;`

where *predicate* is a terminal predicate that must hold eventually whenever the participant has halted exactly here.

For example, consider the following procedure:

```
procedure test [ A ]
{
  send A;
  receive B;
  if A = B
  {
    halt-check ME.A = ME.B;
  }
  halt;
}
```

This defines a simple 'protocol'. Invoking test with a message causes the message to be sent and then another message to be received from the environment. If the two terms message are equal the protocol ends and a terminal predicate equating the values of `A` and `B` must hold. In this example the predicate holds trivially, though. If the messages are not equal the protocol just halts with no security assertions to check.

8.1.4 Terms

Terms are written in the standard notation, i.e.

operator`(`*term*`, ...)`.

For example,

```
cons(A,cons(B,x))
```

is a term built of two operators, two variables and one constant.

8.1.5 Terminal predicates

Terminal predicates are given in the `halt-check` statements. This is the only place to define a terminal predicate. Thus, no 'global' terminal predicate can be defined. This is actually a good thing, because we want to urge the protocol analyst to consider carefully *when* a security assertion should indeed hold.

The terms inside terminal predicates are different from terms in the procedure statements. Here, every variable identifier must be prefixed with a variable of another type that denotes a participant. This is in accordance with the terminal predicates of the formal model. The participant variable `ME` is always bound to the participant itself. A prefixed variable is written as

name-variable`.`*variable*.

For example,

```
ME.A.
```

denotes the variable `A` of the participant itself.

The atomic propositions are

`secret` *term*

denoting that the term *term* must have remained secret, i.e. not known to the environment, and

$term_1$ `=` $term_2$

denoting that the two terms must be equal. Multiple atomic propositions are connected by using

`not` *predicate*

to negate a given predicate;

$predicate_1$ `and` $predicate_2$

to build a conjunction;

$predicate_1$ `or` $predicate_2$

to build a disjunction;

`[`*predicate*`]`

to put a predicate inside parentheses (observe the usage of square brackets, because ordinary parentheses are reserved for the syntax of terms);

`exists` *name-variable predicate*

to bind a participant existentially; and

`for-all` *name-variable predicate*

to bind a participant universally.

For example, the expression

```
[for-all A [ A.VAR = ME.VAR ]] and secret ME.VAR
```

denotes the requirement that all the participants must have the same value bound to the variable `VAR`, and that the value must be secret.

8.1.6 Instantiating Principals

The last thing is to instantiate principals. This is done using the syntax

`principal` *number* *procedure-name* `[` *term*`,` ... `];`

which denotes that *number* instances of a principal that starts by invoking the procedure *procedure-name* with the arguments given inside the square brackets (the terms must be ground terms) should be instantiated.

A complete example is given in Appendix B.

8.2 Formal Grammar

8.2.1 Notation

We use the following notation to make the formal grammar look more concise. Here $\mathscr{X}$ is a variable that will be bound to different grammar symbols.

$$
\begin{array}{rcl}
\textbf{listof}(\ \mathscr{X}) & ::= & \textit{(empty)}\ | \\
 & & \mathscr{X}\ | \\
 & & \mathscr{X}\text{`}\texttt{,}\text{'}\ \textbf{listof}(\ \mathscr{X}) \\
\{\ \mathscr{X}\}^* & ::= & \textit{(empty)}\ | \\
 & & \mathscr{X}\{\ \mathscr{X}\}^* \\
\{\ \mathscr{X}\}^+ & ::= & \mathscr{X}\ | \\
 & & \mathscr{X}\{\ \mathscr{X}\}^*
\end{array}
$$

8.2.2 Lexical structure

$$
\begin{array}{rcl}
\textit{natural} & ::= & \texttt{1}, \ldots, \texttt{9}\ \{\ \texttt{0}, \ldots, \texttt{9}\}^* \\
\textit{identifier} & ::= & \texttt{a}, \ldots, \texttt{z}\ \{\ \texttt{0}, \ldots, \texttt{9}, \texttt{a}, \ldots, \texttt{z}, \texttt{-}\}^* \\
\textit{variable-name} & ::= & \{\ \texttt{A}, \ldots, \texttt{Z}\}^+ \\
\textit{procedure-name} & ::= & \textit{identifier}
\end{array}
$$

As an exception to the above, the following words are reserved and do not parse as identifiers: `and`, `declare-PK-encryption`, `declare-atoms`, `declare-encryption`, `declare-inverse`, `declare-operators`, `declare-public`, `declare-signature-system`, `declare-tuple`, `decrypted-by`, `else`, `exists`, `for-all`, `generate`, `guarded-secret`, `halt`, `halt-check`, `if`, `local`, `not`, `or`, `pragma`, `principal`, `procedure`, `receive`, `return`, `secret`, `send`, `verified-by`, `with-private-key-deriver`, `with-public-key-deriver`, `with-selectors`, `with-signing-key-deriver`, and `with-verification-key-deriver`.

Comments start with `#` and span until the end of the line. Comments are equivalent to nonzero amounts of whitespace.

8.2.3 Terms

$$
\begin{array}{rcl}
\textit{term} & ::= & \textit{identifier} \mid \\
 & & \textit{variable-name} \mid \\
 & & \textit{identifier}\ `\texttt{(}'\ \textbf{listof}(\ \textit{term}\ )\ `\texttt{)}' \\
\textit{ground-term} & ::= & \textit{identifier} \mid \\
 & & \textit{identifier}\ `\texttt{(}'\ \textbf{listof}(\ \textit{ground-term}\ )\ `\texttt{)}' \\
\textit{qualified-term} & ::= & \textit{identifier} \mid \\
 & & \textit{qualified-variable-name} \mid \\
 & & \textit{identifier}\ `\texttt{(}'\ \textbf{listof}(\ \textit{qualified-term}\ )\ `\texttt{)}' \\
\textit{qualified-variable-name} & ::= & \textit{variable-name}\ `\texttt{.}'\ \textit{variable-name}
\end{array}
$$

8.2.4 Statements

halt	::=	'`halt`'
annotated-halt	::=	'`halt-check`' *terminal-predicate*
halting-statement	::=	*halt* |
		annotated-halt
action	::=	'`receive`' *variable-name* |
		'`send`' *variable-name* |
		'`generate`' *variable-name*
assignment	::=	*variable-name* '`:=`' *term*
conditional	::=	'`if`' *variable-name* = *variable-name block* |
		'`if`' *variable-name* = *variable-name block* '`else`' *block*
function-call	::=	*procedure-name* '`[`' **listof**(*term*) '`]`' |
		procedure-name '`[`' **listof**(*term*) '`]`' '`->`' **listof**(*variable-name*)
return	::=	'`return`' **listof**(*term*)
statement	::=	*action* '`;`' |
		assignment '`;`' |
		halting-statement '`;`' |
		function-call '`;`' |
		return '`;`' |
		conditional
block	::=	'{' { *statement* }* '}'

8.2.5 Procedures

procedure-definition	::=	'`procedure`' *procedure-name* '`[`' **listof**(*variable-name*) '`]`'
		autovars block
autovars	::=	*(empty)* |
		'`local`' **listof**(*variable-name*)

8.2.6 Terminal predicates

terminal-predicate	::=	*proposition* |
		‘`not`’ *terminal-predicate* |
		terminal-predicate ‘`and`’ *terminal-predicate* |
		terminal-predicate ‘`or`’ *terminal-predicate* |
		‘`[`’ *terminal-predicate* ‘`]`’ |
		qualified-terminal-predicate
qualified-terminal-predicate	::=	‘`exists`’ *variable-name terminal-predicate* |
		‘`for-all`’ *variable-name terminal-predicate*
proposition	::=	*equality-proposition* |
		nondisclosure-proposition
equality-proposition	::=	*qualified-term* ‘`=`’ *qualified-term*
nondisclosure-proposition	::=	‘`secret`’ *qualified-term*

8.2.7 Cryptosystem definitions

operator-declarations ::= ‘`declare-operators`’ **listof**(*identifier*) ‘;’
atom-declarations ::= ‘`declare-atoms`’ **listof**(*identifier*) ‘;’
public-declarations ::= ‘`declare-public`’ **listof**(*ground-term*) ‘;’
property-definition ::= *inv-property* |
symenc-property |
pubenc-property |
tuple-property |
signature-property
inv-property ::= ‘`declare-inverse`’ *identifier* ‘`to`’ *identifier* ‘;’
symenc-property ::= ‘`declare-encryption`’ *identifier*
‘`decrypted-by`’ *identifier* ‘;’
symenc-property ::= ‘`declare-PK-encryption`’ *identifier*
‘`decrypted-by`’ *identifier*
‘`with-public-key-deriver`’ *identifier*
‘`with-private-key-deriver`’ *identifier* ‘;’
signature-property ::= ‘`declare-signature-system`’ *identifier*
‘`verified-by`’ *identifier*
‘`with-signing-key-deriver`’ *identifier*
‘`with-verification-key-deriver`’ *identifier* ‘;’
tuple-property ::= ‘`declare-tuple`’ *identifier*
‘`with-selectors`’ **listof**(*identifier*) ‘;’
declaration ::= *operator-declarations* |
atom-declarations |
public-declarations |
property-definition

8.2.8 Pragmas

pragma ::= ‘`pragma`’ *pragma-body* ‘;’
pragma-body ::= ‘`enable-output`’ *output-path* |
‘`disable-output`’ *output-path* |
‘`enable`’ *generic-option* |
‘`disable`’ *generic-option*
output-path ::= *identifier* { ‘`:`’ *identifier* }*
generic-option ::= *identifier*

8.2.9 Principals

$$\begin{array}{rcl} \textit{principal} & ::= & \text{`}\texttt{principal}\text{'}\ \textit{natural}\ \textit{entry}\ \text{`}\texttt{;}\text{'} \\ \textit{entry} & ::= & \textit{procedure-name}\ \text{`}\texttt{[}\text{'}\ \textbf{listof}(\ \textit{ground-term}\ )\ \text{`}\texttt{]}\text{'} \end{array}$$

8.2.10 Input

$$\textit{input} \ ::= \ \{\ \textit{pragma}\ \}^*\ \{\ \textit{declaration}\ \}^*\ \{\ \textit{procedure-definition}\ \}^+\ \{\ \textit{principal}\ \}^+$$

8.3 Semantics

In this section we give the semantics of the input language. This is done in principle with the aids of denotational semantics [21]. However, we use use denotational semantics only as a convenient tool for describing the semantics to the level of rigor we see suitable. Complete semantics are not given formally; many less interesting parts are handled informally. We trust the reader's intuition.

Concerning the procedures, the basic idea of that the variables that appear as procedure arguments or local variables are mapped to freshly generated new state variables every time the procedure is called. All procedure calls are 'inlined', because there is no procedure construction in the formal model.

We give denotations for procedures. A procedure's denotation is a function that returns a *partial program* when it is invoked with a list of terms and a list of variables. The list of terms corresponds to the parameters the procedure is called with. The list of variables is where the return values of the procedure call should be stored.

A partial program is an element of $\mathcal{P}_*$, however with the addition that we allow $\square$ as a special type of terminal program. Thus, e.g. $\urcorner\ v.\square$ is a partial program.

Partial programs can be joined. This is accomplished by the function join that maps two partial programs into a new partial program, defined as

$$\begin{aligned}
\mathrm{join}(\mathbf{halt}_n, x) &= \mathbf{halt}_n \\
\mathrm{join}(\Box, x) &= x \\
\mathrm{join}(\Rsh v.p, x) &= \Rsh v.\mathrm{join}(p, x) \\
\mathrm{join}(\Lsh v.p, x) &= \Lsh v.\mathrm{join}(p, x) \\
\mathrm{join}(\uparrow v.p, x) &= \uparrow v.\mathrm{join}(p, x) \\
\mathrm{join}(v \leftarrow t.p, x) &= v \leftarrow t.\mathrm{join}(p, x) \\
\mathrm{join}((v = v')?p : p', x) &= (v = v')?\mathrm{join}(p, x) : \mathrm{join}(p', x)
\end{aligned}$$

Intuitively, $\mathrm{join}(p, x)$ denotes a program that is obtained by appending x to p, i.e. first executing the actions of p and then those of x.

The following notation will be used for the denotational semantics: $\langle s_1, \dots, s_k\rangle$ is a sequence consisting of the elements $s_1, \dots, s_k$. $s \S t$ is the concatenation of the two sequences s and t. $s \downarrow n$ denotes the nth element of s (the first element is $s \downarrow 1$). $s \dagger n$ denotes the sequence that is obtained by dropping out the first n elements from s. $\#s$ denotes the length of the sequence s.

8.3.1 Abstract syntax

The abstract syntax of procedure definitions is shown in Figure 8.1.

For **proc**, the first V is the list of formal arguments and the second V is the list of local variables. Both may be empty. For **if**, the two V's are the variables to compare, the first C is the consequent and the second C the alternative.

8.3.2 Domains

The following domains are used:

P	Partial programs
E	Errors
$\mathbf{T}_{\mathcal{S}}(\mathcal{X})$	Terms
$\mathcal{V}$	State variables
$\mathbb{N}$	Natural numbers

$T \in \text{Term}$	terms
$V \in \text{Var}$	state variable names
$C \in \text{Cla}$	clauses
Proc	procedures
$N \in \text{Nam}$	procedure names

$$
\begin{array}{rcl}
\mathit{Proc} & ::= & \mathbf{proc}(N)(V^*)(V^*)(C) \\
\mathit{Cla} & ::= & \mathbf{seq}(C^*) \mid \\
& & \mathbf{if}(V)(V)(C)(C) \mid \\
& & \mathbf{call}(N)(T^*)(V^*) \mid \\
& & \mathbf{send}(V) \mid \\
& & \mathbf{recv}(V) \mid \\
& & \mathbf{gen}(V) \mid \\
& & \mathbf{let}(V)(T) \mid \\
& & \mathbf{halt} \mid \\
& & \mathbf{return}(T^*)
\end{array}
$$

Figure 8.1 The abstract syntax of procedure definitions.

$$
\begin{aligned}
\mathscr{S} &: \mathrm{Proc} \mapsto \mathrm{Term}^* \mapsto \mathrm{Var}^* \mapsto P \\
\mathscr{P} &: \mathrm{Cla} \mapsto (\mathrm{Var} \mapsto \mathrm{Var}) \mapsto \mathrm{Var}^* \mapsto P \\
\mathscr{I} &: \mathrm{Var} \mapsto \mathcal{V} \\
\mathscr{T} &: \mathrm{Term} \mapsto \mathbf{T}_{\mathcal{S}}(\mathcal{X}) \\
\mathrm{makecall} &: P \mapsto \mathrm{Term}^* \mapsto \mathrm{Var}^* \mapsto \mathrm{Var}^* \mapsto \mathrm{Var}^* \mapsto P \\
\mathrm{join} &: P \mapsto P \mapsto P \\
\mathrm{uniqueint} &: \mathbb{N} \\
\mathrm{maketemps} &: \mathbb{N} \mapsto \mathrm{Var}^* \\
\mathrm{map} &: \mathrm{Var}^* \mapsto \mathrm{Var}^* \mapsto (\mathrm{Var}^* \mapsto \mathrm{Var}^*) \\
\mathrm{error} &: E \mapsto \bot \\
\mathrm{bindret} &: \mathrm{Var}^* \mapsto \mathrm{Term}^* \mapsto P \\
\mathrm{procbyname} &: N \mapsto \mathrm{Term}^* \mapsto \mathrm{Var}^* \mapsto P
\end{aligned}
$$

Figure 8.2 The signatures of the semantic functions.

8.3.3 Semantic Functions

The signatures of the semantic functions we need are shown in Figure 8.2.

$\mathscr{S}$ is the function giving a denotation to a procedure definition. $\mathscr{P}$ gives a denotation to a clause, i.e. a program fragment. $\mathscr{I}$ and $\mathscr{T}$ are uninteresting functions that map variable names and terms from the language level to the formal model. Their definitions are deliberately omitted. makecall is defined below. join was already defined. uniqueint is a magic function returning a distinct natural number every time it is invoked. We deliberately omitted the argument that would have needed to make this function a standard one. Similarly, maketemps is a function that generates a required number of new unique state variable names. map creates a function from two sequences of variable names; a definition is given below. makecall and bindret are auxiliary functions that are used when giving denotation for procedure calls; definitions are given below. error is a function that does something with an error message. error always maps to $\bot$. That is, an erronous program denotes $\bot$.

On the language level there is a sequence of procedure definitions. Moving this to the level of semantics would require some kind of store where the actual procedures would be associated with their names. We have not modeled this store. Therefore we use a function procbyname that returns the procedure

$$
\begin{aligned}
\mathscr{S}[\![\mathbf{proc}(N)(V_1^*)(V_2^*)(C)]\!] &= \lambda\alpha\rho.\text{makecall}\mathscr{P}[\![C]\!]\alpha\rho V_1^* V_2^* \\
\mathscr{P}[\![\mathbf{seq}(CC^*)]\!] &= \lambda\pi\rho.\text{join } \mathscr{P}[\![C]\!]\pi\rho\mathscr{P}[\![\mathbf{seq}(C^*)]\!]\pi\rho \\
\mathscr{P}[\![\mathbf{seq}()]\!] &= \square \\
\mathscr{P}[\![\mathbf{call}(N)(T^*)(V^*)]\!] &= \lambda\pi\rho.\text{procbyname } N(\pi T^*)(\pi V^*) \\
\mathscr{P}[\![\mathbf{send}(V)]\!] &= \lambda\pi\rho.\ \Lsh\ \mathscr{I}[\![\pi V]\!].\square \\
\mathscr{P}[\![\mathbf{recv}(V)]\!] &= \lambda\pi\rho.\ \Rsh\ \mathscr{I}[\![\pi V]\!].\square \\
\mathscr{P}[\![\mathbf{gen}(V)]\!] &= \lambda\pi\rho.\uparrow \mathscr{I}[\![\pi V]\!].\square \\
\mathscr{P}[\![\mathbf{let}(V)(T)]\!] &= \lambda\pi\rho.\mathscr{I}[\![\pi V]\!] \leftarrow \mathscr{T}[\![\pi V]\!].\square \\
\mathscr{P}[\![\mathbf{halt}]\!] &= (\lambda i.\mathbf{halt}_i)(\text{uniqueint}) \\
\mathscr{P}[\![\mathbf{if}(V_1)(V_2)(C_1)(C_2) &= \lambda\pi\rho.(\mathscr{I}[\![\pi V_1]\!] = \mathscr{I}[\![\pi V_2]\!])?\mathscr{P}[\![C_1]\!]\pi\rho : \mathscr{P}[\![C_2]\!]\pi\rho \\
\mathscr{P}[\![\mathbf{return}(T^*)]\!] &= \lambda\pi\rho.\#\rho = \#T^* \rightarrow \text{bindret } \rho(\pi T^*), \\
&\qquad \text{error `wrong number of returned values'}
\end{aligned}
$$

Figure 8.3 The semantic functions.

denoted by a given name but do not give an explicit definition.

The semantic functions are shown in Figure 8.3.

The auxiliary functions are shown in Figure 8.4.

8.4 Technical Details

We implemented the actual parser for the language as a hand-crafted recursive-descent top-down parser. For the lexical parsing, however, we used Danny Dubé's SILex system [24].

We wrote the parser partly by hand mostly because of personal inclination; parser generators exist for Scheme.

$$\begin{aligned}
\text{makecall} = \lambda p\alpha\rho x\ell.\#\alpha \neq \#x \rightarrow \text{error `wrong number of args'},\\
(\lambda\pi'.\text{join}(\mathscr{P}[\![\mathbf{seq}(\mathbf{let}(x \downarrow 1)(\alpha \downarrow 1) \cdots\\
\mathbf{let}(x \downarrow \#x)(\alpha \downarrow \#x))]\!](\pi \circ \pi')\bot)\\
(p\pi'\rho))\\
((\lambda t.\text{map } (x \S \ell)t)(\text{maketemps } (\#x + \#\ell)))
\end{aligned}$$

$$\text{map} = \lambda st.s = \langle\rangle \rightarrow \bot, (\text{map}(s \dagger 1)(t \dagger 1))[s \downarrow 1 \mapsto t \downarrow 1]$$

$$\text{bindret} = \lambda\rho t.\rho = \langle\rangle \rightarrow \square, \rho \downarrow 1 \leftarrow t \downarrow 1.\text{bindret}(\rho \dagger 1)(t \dagger 1)$$

Figure 8.4 The auxiliary functions.

Chapter 9

Implementation Details

Some details of the implementation are discussed, namely certain modifications in the algorithms, the options supported by the tool and an extension that can be used to give participant's human-understandable names.

There are certain implementation details that should be mentioned. These details are the subject of this chapter.

9.1 Changes in the Algorithms

Some changes were made in the algorithms. First, the actual symmetry method was not implemented (although symbolic state subsumption was).

Second, a very important extra optimization was added to the search driver. Namely, those maximal symbolic steps that end in the corresponding participant halting typically do not affect the environment's knowledge. When the state space is generated, such steps can be ignored as long as there are some symbolic steps available that cause the knowledge to increase.

Actually, the step computation happens so that symbolic steps end after send actions that change the environment's knowledge instead of receive actions. This is not a major modification, but it helps to get rid of the extra parameter $\hat{g}_0$ in function STEP (see Chapter 6).

Those variables in the principals' valuations that are no longer used by their processes nor the terminal predicate can be removed from the valuation. This is useful because it makes the symbolic state subsumption relation work better, as the values of irrelevant variables need not to be unified.

9.2 Options

The following options are supported by the implementation: `filter-early`, `aggressive-filtering`, `ignore-nonactive`, `short-steps`, and `symmetry-reductions`.

`filter-early` enables terminal predicate rewriting. The name comes from the fact that rewriting the terminal predicate on the fly typically causes many branches of the state space to be pruned quite early.

`aggressive-filtering` is an option modifying terminal predicate rewriting. If `aggressive-filtering` is specified, the terminal predicates are rewritten even inside steps. Otherwise they are rewritten only between symbolic steps. The latter case corresponds to the algorithm presented in Chapter 6.

`ignore-nonactive` enables the extra optimization mentioned in the previous subsection. The name comes from the fact that we ignore steps that actually do not change the environment.

`short-steps` modifies the concept of symbolic steps so that a step ends before any receiving action instead of only those for which the environment's knowledge has actually increased.

`symmetry-reductions` enables the reduction based on the symbolic quasi-symmetry reduction. Because the actual symmetries are not implemented this corresponds to the symbolic state subsumption relation.

9.3 Participant Names

When the analyser finds an attack, a history that leads to the insecure state found is displayed. For the purposes of this displaying, the state variable with the name `identity` has a special purpose. If the variable is set, then the term it is assigned to is regarded as the principal's identity and displayed in the history, replacing the principal's formal name that is an element of $\mathscr{X}_P$ and that will be displayed otherwise. However, the variable has no other specific meaning. It is just a normal state variable.

Chapter 10

Example

An example is given, using the Needham-Schröder Public-Key Protocol as a case. The tool reproduces an attack originally due to Gavin Lowe in a short time and outputs it in a user-friendly format.

10.1 The Attack Against the Needham-Schröder Public-Key Protocol

In this chapter we present an example about analyzing protocols by using the protocol analyzer implemented. Our example focuses on the Needham-Schröder Public-Key Protocol. This authentication protocol is very old; it was invented in the late 70's [71]. It can be represented as follows [44], using the customary notion for authentication protocols:

$$
\begin{array}{llllll}
(1) & A & \longrightarrow & S & : & A, B \\
(2) & S & \longrightarrow & A & : & \{K_b, B\}_{K_s^{-1}} \\
(3) & A & \longrightarrow & B & : & \{N_a, A\}_{K_b} \\
(4) & B & \longrightarrow & S & : & B, A \\
(5) & S & \longrightarrow & B & : & \{K_a, A\}_{K_a^{-1}} \\
(6) & B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\
(7) & A & \longrightarrow & B & : & \{N_b\}_{K_b}
\end{array}
$$

Here $\{x\}_y$ denotes x encrypted with y. K_a is A's public key and K_a^{-1} the corresponding private key.

In this protocol there are three parties. Alice, A, is the *initiator* who seeks to establish a session with Bob, B, the *responder*. Basically the protocol is a public-key protocol. However, it is well known that public keys need to be certified in some way in order to prevent man-in-the-middle attacks. In this protocol the certification authority is explicitly presented. S is an authentication server that issues certificates about public keys belonging to certain parties.

The "actual" authentication protocol between Alice and Bob does not however require S to exist. In the full protocol as shown above, the messages 1,2,4 and 5 are certificate requests and transfers and the messages 3, 6 and 7 are the actual authentication.

The certificate handling is pretty trivial. The interesting part of the protocol are the actual authentication messages. Therefore we can restrict our attention to the following protocol:

$$\begin{array}{llcll} (3) & A & \longrightarrow & B & : \{N_a, A\}_{K_b} \\ (6) & B & \longrightarrow & A & : \{N_a, N_b\}_{K_a} \\ (7) & A & \longrightarrow & B & : \{N_b\}_{K_b} \end{array}$$

This protocol contains a design mistake that was discovered by Gavin Lowe [44]. The mistake is that the sixth message does not explicitly mention the name of the sender, i.e. Bob. This mistake can be turned into a severe attack that Lowe also demonstrated.

In this attack there are two interleaved protocol runs. In the first run, Alice tries to establish a connection with a malicious party; let us name this party Igor the Intruder. In the second run Igor is able to impersonate Alice to Bob, thus making Bob believe that he is communicating with Alice. The attack goes as follows:

$$\begin{array}{llcll} (1.3) & A & \longrightarrow & I & : \{N_a, A\}_{K_i} \\ (2.3) & I & \longrightarrow & B & : \{N_a, A\}_{K_b} \\ (2.6) & B & \longrightarrow & I & : \{N_a, N_b\}_{K_a} \\ (1.6) & I & \longrightarrow & A & : \{N_a, N_b\}_{K_a} \\ (1.7) & A & \longrightarrow & I & : \{N_b\}_{K_i} \\ (2.7) & I & \longrightarrow & B & : \{N_b\}_{K_b} \end{array}$$

```
[alice] receives igor
[alice] sends rsa-enc(rsa-dpub(igor-keym), cons(N2.0, alice))
[bob] receives rsa-enc(rsa-dpub(bob-keym), cons(N2.0, alice))
[bob] sends rsa-enc(rsa-dpub(alice-keym), cons(N2.0, N1.0))
[alice] receives rsa-enc(rsa-dpub(alice-keym), cons(N2.0, N1.0))
[alice] sends rsa-enc(rsa-dpub(igor-keym), N1.0)
[bob] receives rsa-enc(rsa-dpub(bob-keym), N1.0)
```

Figure 10.1 Example output from the analysis tool.

From the Bob's viewpoint, the following protocol has taken place:

$$\begin{array}{lcccl} (2.3) & A & \longrightarrow & B & : & \{N_a, A\}_{K_b} \\ (2.6) & B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\ (2.7) & A & \longrightarrow & B & : & \{N_b\}_{K_b} \end{array}$$

Thus, Bob believes he is communicating with Alice.

10.2 Reproducing the Attack

In order to demonstrate the analyzer in action, we reproduce the attack by using the analyzer. The input file that models the protocol is shown in Appendix B.

Given the input file, the analyzer finds the attack in a few seconds, giving the output shown in Figure 10.1. The nice format from which the attack can be unambiguously and easily observed should be appreciated.

Chapter 11

Conclusions

A real winner does not turn back to look at his own accomplishments. —Anonymous

11.1 Summary

We have presented a new formal analysis method for security protocols in this thesis. The method is based on symbolic state space enumeration. In addition to the basic method, we have augmented it with the concept of a symbolic step transition, a symbolic symmetry method and a method for detecting symbolic state subsumption relations.

We have modeled cryptography using a parameterized rewrite system that can be used to model many kinds of cryptographic primitives, including hash functions, symmetric and idempotent encryption, public-key encryption, signature systems and revertible transformations.

Typically, models involving rewrite systems are implemented by using variants of the narrowing algorithm [20]. However, our method does not need to use narrowing because of a novel, alternative approach bound tightly to the whole search process. The method is based on the concept of a basis that removes the need to consider the rewrite systems in some parts of the formal machinery.

The presented method is relatively fast. More importantly, it is the first method known to us that enumerates the complete infinite state space without encountering decidability problems nor requiring user interaction.

We have also described an implementation of the method as an end-user tool for security protocol analysis. The tool introduces a higher-level programming language that can be used to describe cryptographic protocols in a concise and intuitive way.

11.2 Critical Discussion and Suggestions for Future Research

Although the results we have obtained are useful and have potential for further development, some shortcomings exist that must be mentioned.

The presentation of the work is a bit too compact and the theory is not polished at all. A technical problem with the work was the fact that a more thorough presentation would have required still more pages, which would have been unbearable. The immature nature of the theory must be tolerated; it takes many "iterations" for a theory to achieve its final form.

Our method does not currently support associative nor commutative operators. It is a well-known fact that such operators always cause more or less problems. Extending the methodology to support AC-operators is a future challenge. Initial considerations in this direction has revealed some deep technical questions that need to be answered. It is conceivable that the solver needs to go through a major revision before AC-operators can be supported.

11.3 Final Remarks

This thesis is a demonstration of the fact that symbolic state space search can be a very feasible method for the formal analysis of various concurrent systems.[1]

[1] Our thesis also gives the hope that customized symbolic state representations and methods for manipulating them can be a cornucopia of academic publications.

Bibliography

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996. First in Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy, pages 122–136. Also as a research report.

[2] J. Alves-Foss. The use of belief logics in the presence of causal consistency attacks. In *Proceedings of the 20th National Information Systems Conference*, pages 406–417, 1997.

[3] R. Anderson and R. Needham. Robustness principles for public key protocols. In *Advances in Cryptology — Crypto '95*, pages 236–247, 1995.

[4] G. Bella and L. C. Paulson. Using Isabelle to prove properties of the kerberos authentication system. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 1997. The proceedings is available only in electronic form.

[5] G. Bella and L. C. Paulson. Mechanising BAN Kerberos by the inductive method. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification — CAV '98*, number 1427 in Lecture Notes in Computer Science, pages 416–427. Springer-Verlag, 1998.

[6] P. Bieber. Methods for reasoning about cryptographic protocols. In *Proceedings of the Cambridge Formal Methods Workshop*, 1993.

[7] C. Boyd. A formal framework for authentication. In *Proceedings of the Second European Symposium on Research in Computer Security — ESORICS 92*, pages 273–292, 1992.

[8] C. Boyd and W. Mao. Designing secure key exchange protocols. In *Proceedings of the Third European Symposium on Research in Computer Security — ESORICS 94*, pages 93–105, 1994.

[9] C. Boyd and W. Mao. On limitations of logical analysis of cryptographic protocols (on a limitation of BAN logic). In *Advances in Cryptology — Eurocrypt '93*, pages 240–247. Springer-Verlag, 1994. Was refuted by van Oorschot in the rump session of the conference where the paper was presented.

[10] S. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop IX*. IEEE Computer Society Press, 1996.

[11] S. H. Bracking. Evaluating and improving protocol analysis by automatic proof. In *Proceedings of the Computer Security Foundations Workshop XI*, pages 138–152, 1998.

[12] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proceedings of the Royal Society, Series A, 426, 1871*, Dec. 1989. Appeared also as SRC Research Report 39 (February 1989) and, in a shortened form, in ACM Transactions on Computer Systems 8(1):18–36 (February 1990).

[13] M. Burrows, M. Abadi, and R. Needham. The scope of a logic of authentication. In *Distributed Computing and Cryptography: Proceedings of a DIMACS Workshop*, pages 119–126, Oct. 1989.

[14] M. Burrows, M. Abadi, and R. Needham. Rejoinder to nessett. *ACM Operating Systems Review*, 20(2):39–40, Apr. 1990.

[15] L. Buttyán, S. Staamann, and U. Wilhelm. A simple logic for authentication protocol design. In *Proceedings of the Computer Security Foundations Workshop XI*, pages 153–162, 1998.

[16] E. A. Campbell, E. Safavi-Naini, and P. A. Pleasants. Partial belief and probabilistic reasoning in the analysis of secure protocols. In *Proceedings of the Computer Security Foundations Workshop V*, pages 92–110. IEEE Computer Society Press, 1992.

[17] J. A. Clark and J. Jacob. A survey of authentication protocol literature. manuscript, Aug. 1996.

[18] Z. Dang and R. Kemmerer. Using the ASTRAL model checker for cryptographic protocol analysis. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 1997. The proceedings is available only in electronic form.

[19] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, Aug. 1981.

[20] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier Science Publishers B.V., 1990.

[21] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Denotational Semantics, pages 575–631. Elsevier Science Publishers B.V., 1990.

[22] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, Mar. 1983.

[23] J. Domingo-Ferrer. Probabilistic authentication analysis. In *CARDIS '94 — Proceedings of the First Smart Card Research and Applications Conference*, pages 49–60, Lille, France, Oct. 1994.

[24] D. Dubé. Silex — a Scheme implementation of Lex.

[25] Formal Systems (Europe) Ltd. Failures-divergence refinement — FDR 2 — user manual, 1997. Available via URL `ftp://ftp.comlab.ox.ac.uk/pub/Packages/FDR`.

[26] F. Germeau and G. Leduc. Model-based design and verification of security protocols using LOTOS. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 1997. The proceedings is available only in electronic form.

[27] D. Gollmann. What do we mean by entity authentication? In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 46–54. IEEE Computer Society Press, 1996.

[28] L. Gong. Efficient network authentication protocols: Lower bounds and optimal implementations. *Distributed Computing*, 9(3):131–145, 1995.

[29] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society Press, May 1990.

[30] L. Gong and P. F. Syverson. Fail-stop protocols: An approach to designing secure protocols. In *Proceedings of DCCA5*. Springer-Verlag, 1995.

[31] S. Gritzalis, N. Nikitakos, and P. Georgiadis. Formal methods for the analysis and design of cryptographic protocols: A state-of-the-art review. In *Third IFIP TC6/TC11 Working Conference on Communications and Multimedia Security*, pages 119–132, 1997.

[32] R. C. Hauser. Verification and modelling of authentication protocols. In *Proceedings of the Second European Symposium on Research in Computer Security — ESORICS 92*, pages 141–154. Springer-Verlag, 1992.

[33] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[34] A. Huima. Security flaws in popular security software: Lessons learned from problems in SSH-1.2.17 and older. In *Proceedings of the Second Nordic Workshop on Secure Computer Systems (NORDSEC '97)*, Nov. 1997. The version in the workshop proceedings contains a mistake in the treatment of the authentication procedure; we refer to the corrected version (available from the author).

[35] IEEE. IEEE standard 1179-1990. IEEE standard for the Scheme programming language, 1991.

[36] R. Kailar and V. D. Gligor. On belief evolution in authentication protocols. In *Proceedings of the Computer Security Foundations Workshop IV*, pages 102–116. IEEE Computer Society Press, June 1991.

[37] R. Kelsey, W. Clinger, and J. Rees, editors. *Revised^5 Report on the Algorithmic Language Scheme*, 1998.

[38] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[39] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 2nd edition, 1988.

[40] V. Kessler and G. Wedel. AUTLOG — an advanced logic of authentication. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 90–99. IEEE Computer Society Press, 1994.

[41] D. King. Proofpower implementation of BAN logic. In *Proceedings of the Cambridge Formal Methods Workshop*, 1993.

[42] X. Lai and J. Massey. A proposal for a new block encryption standard. In I. Damgård, editor, *Proc. EUROCRYPT 90*, pages 389–404. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 473.

[43] X. Lai and J. Massey. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Proc. EUROCRYPT 91*, number 547 in Lecture Notes in Computer Science, pages 17–38. Springer, 1991.

[44] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[45] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Margaria and Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 147–166. Springer-Verlag, 1996.

[46] G. Lowe. Some new attacks upon security protocols. In *Proceedings of the Computer Security Foundations Workshop IX*. IEEE Computer Society Press, 1996.

[47] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of the Computer Security Foundations Workshop X*. IEEE Computer Society Press, 1997.

[48] G. Lowe. SPLICE/AS: A case study in using CSP to detect errors in security protocols. *Distributed Computing*, 1998. (submitted).

[49] G. Lowe. Towards a completeness result for model checking of security protocols (extended abstract). In *Proceedings of the Computer Security Foundations Workshop XI*. IEEE Computer Society Press, 1998.

[50] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997. Preliminary version published as Department of Mathematics and Computer Science Technical Report 1996/34, University of Leicester, 1996.

[51] W. Mao. An augmentation of BAN-like logics. In *Proceedings of the Computer Security Foundations Workshop VIII*, pages 44–56. IEEE Computer Society Press, June 1995.

[52] W. Mao and C. Boyd. On strengthening authentication protocols to foil cryptanalysis. In *Proceedings of the Third European Symposium on Research in Computer Security — ESORICS 94*, pages 193–204, 1994.

[53] W. Mao and C. Boyd. Methodical use of cryptographic transformations in authentication protocols. In *IEE Proceedings, Comput. Digit. Tech*, volume 142, pages 272–278, July 1995.

[54] A. Mathuria, R. Safavi-Naini, and P. Nickolas. Exploring minimal BAN logic proofs of authentication protocols. In *Tenth International Information Processing Conference — Security '94 (IFIP-SEC '94)*, May 1994.

[55] A. Mathuria, R. Safavi-Naini, and P. Nickolas. On the automation of GNY-logic. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 370–379, 1995.

[56] C. A. Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 84–89. IEEE Computer Society Press, 1994.

[57] C. A. Meadows. The NRL protocol analyzer: An overview. In *Proceedings of the 2nd Conference on the Practical Applications of Prolog*, Apr. 1994.

[58] C. A. Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology — Asiacrypt '94*, number 917 in Lecture Notes in Computer Science, pages 133–150. Springer-Verlag, 1995.

[59] C. A. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proceedings of the Fourth European Symposium on Research in Computer Security — ESORICS 96*. Springer-Verlag, 1996.

[60] C. A. Meadows. Language generation and verification in the NRL protocol analyzer. In *Proceedings of the Computer Security Foundations Workshop IX*. IEEE Computer Society Press, 1996.

[61] C. A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[62] J. K. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 251–260. IEEE Computer Society Press, 1995.

[63] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, Feb. 1987.

[64] J. C. Mitchell. Finite-state analysis of security protocols. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification — CAV '98*, number 1427 in Lecture Notes in Computer Science, pages 71–76. Springer-Verlag, 1998.

[65] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, 1997.

[66] J. C. Mitchell, V. Schamtikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proceedings of the Sevent USENIX Security Symposium*, pages 201–215, 1998.

[67] J. C. Mitchell, V. Slunatikov, and U. Stern. Finite-state analysis of SSL 3.0 and related protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 1997. The proceedings is available only in electronic form.

[68] National Bureau of Standards. Announcing the data encryption standard. Technical Report FIPS Publication 46, National Bureau of Standards, Jan. 1977.

[69] National Institute of Standards and Technology (NIST). *FIPS Publication 180: Secure Hash Standard (SHS)*, May 11, 1993.

[70] National Institute of Standards and Technology (NIST). *Announcement of Weakness in the Secure Hash Standard (SHS)*, May 1994.

[71] R. M. Needham and M. D. Schröder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, Dec. 1978.

[72] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, Inc., 1994.

[73] L. C. Paulson. Inductive analysis of the internet protocol TLS. Report 440, University of Cambridge, Computer Laboratory, 1997.

[74] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *Proceedings of the Computer Security Foundations Workshop X*, pages 84–95. IEEE Computer Society Press, June 1997. In the conference an abridged version was published; we refer to the full paper.

[75] L. C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Report 413, University of Cambridge, Computer Laboratory, 1997.

[76] L. C. Paulson. Proving properties of security protocols by induction. In *Proceedings of the Computer Security Foundations Workshop X*, pages 70–83. IEEE Computer Society Press, 1997.

[77] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare.* Prentice-Hall, 1994.

[78] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the Computer Security Foundations Workshop VIII*, pages 98–107, 1995.

[79] A. D. Rubin and P. Honeyman. Formal methods for the analysis of authentication protocols. Technical Report 93–7, CITI, Nov. 1993.

[80] S. Schneider. Security properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[81] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Proceedings of the Computer Security Foundations Workshop XI*. IEEE Computer Society Press, 1998.

[82] E. Snekkenes. Exploring the BAN approach to protocol analysis. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 171–181. IEEE Computer Society Press, 1991.

[83] S. Stubblebine and C. A. Meadows. On searching for known and chosen cipher pairs using the NRL protocol analyzer. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Sept. 1997. The proceedings is available only in electronic form.

[84] P. Syverson and C. Meadows. A logical language for specifying cryptographic protocol requirements. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE Computer Society Press, May 1993.

[85] P. Syverson and P. C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28. IEEE Computer Society Press, 1994.

[86] P. F. Syverson. Adding time to a logic of authentication. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 97–106. ACM Press, Nov. 1993.

[87] P. F. Syverson. Limitations on design principles for public key protocols. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 62–73. IEEE Computer Society Press, 1996.

[88] P. F. Syverson and C. A. Meadows. Formal requirements for key distribution protocols. In *Advances in Cryptology — Eurocrypt '94*. Springer-Verlag, 1995.

[89] P. C. van Oorschot. Extending cryptographic logics of belief to key agreement protocols (extended abstract). In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 232–243, 1993.

[90] P. C. van Oorschot. An alternate explanation of two BAN-logic "failures". In *Advances in Cryptology — Eurocrypt '93*, pages 443–447. Springer-Verlag, 1994.

[91] G. Wedel and V. Kessler. Formal semantics for authentication logics. In *Proceedings of the Fourth European Symposium on Research in Computer Security — ESORICS 96*, pages 219–241, 1996.

[92] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. *ACM Operating Systems Review*, 28(3):24–37, July 1994.

[93] S. Xu, G. Zhang, and H. Zhu. On the properties of cryptographic protocols and the weaknesses of the BAN-like logics. *ACM Operating Systems Review*, 31(4):12–23, Oct. 1997.

[94] D. Zheng, J. B. Tian, and Y. M. Wang. A note on the augmenting of BAN-like logics. In *Proceedings of the 5th Chinese Conference on Cryptology*, pages 123–126, 1998.

Index

Appendix A

Complexity results

The analysis problem of this thesis is shown to be **NP***-hard, and the unbounded version of it is shown to be undecidable. Both results are established by showing that the formalism for cryptographic protocols used in this thesis can be used to describe a protocol that simulates an arbitrary nondeterministic Turing machine step by step.*

Theorem A.1 *Let Σ be fixed, and let Q denote the countable set of all nondeterministic Turing machines operating on the alphabet Σ. Then there exists a transformation T computable in polynomial time $(Q \times \Sigma^*) \mapsto \mathcal{P}$ such that if $T(\mathscr{M}, w) = p$, then there exists a set of participant names $\mathcal{N}$, $|\mathcal{N}| = k$ such that for the mapping $\mathscr{L} : (\forall n)[\mathscr{L}(n) = \langle p, \epsilon\rangle]$, it holds that $\langle \mathscr{L}, \emptyset, \mathbb{N}\rangle \to^* \langle \mathscr{L}', M, N\rangle$, where $\dagger \in M^*$, exactly if there is an accepting computation of less than k steps for $\mathscr{M}$ on the input w.*

Proof. The idea of the proof is to construct a program p working roughly as follows: First the program receives a message, which should be an encrypted Turing machine state. Then the program receives another message that represents a nondeterministic choice to be made. Then the program computes the successor state of the received state, using the nondeterministic choice designated, and sends it to the environment encrypted. At any moment, the environment contains a set of encrypted Turing machine states that are all reachable from some originally given initial state. Then, if the program receives a state that represents the end of an accepting computation, the program sends the distinguished message "†" to the environment. This message is otherwise not mentioned anywhere.

Because then every run of the program p corresponds to at most one step of execution for the Turing machine simulated, k runs can simulate at most k steps. Because the states are encrypted, the intruder cannot "invent" states that are not constructed by the honest participants running p. This completes the proof.

The details are omitted as they can be relatively straightforwardly constructed. $\square$

The following two corollaries are the important complexity results of this appendix.

Corollary A.2 *The analysis problem of this thesis is* **NP***-hard.* $\square$

Of course, it would be more interesting to know the exact position of our problem in the formal complexity hierarchy [72]. It seems that the problem belongs to **NP**, and is thus **NP**-complete. However, a formal proof of this proposition would be outside the scope of this thesis.

The **NP**-hardness result in itself is enough to justify the fact that the algorithms we developed may behave exponentially. Without a major breakthrough in the Computer Science community, all generic solutions to our problem will have superpolynomial complexities.

Corollary A.3 *The unbounded version of the analysis problem, i.e. whether or not a protocol is secure for an unbounded number of participants, is undecidable.* $\square$

This corollary, as mentioned in the introduction, justifies our decision to confine to a bounded number of participants.

Appendix B

The Needham-Schröder Public-Key Protocol

The following listing was used to model the Needham-Schröder Public-Key Protocol:

```
pragma enable   filter-early
pragma enable   ignore-nonactive
pragma disable  short-steps
pragma disable  symmetry-reductions

declare-operators rsa-enc, rsa-dpub, rsa-dpriv;
declare-operators car, cons, cdr;

declare-PK-encryption          rsa-enc
  decrypted-by                 rsa-enc
  with-public-key-deriver      rsa-dpub
  with-private-key-deriver     rsa-dpriv;

declare-tuple                  cons
  with-selectors               car, cdr;

declare-atoms alice, bob, igor, alice-keym, bob-keym, igor-keym, good, bad;

declare-public alice, bob, igor;
declare-public rsa-dpub(alice-keym);
declare-public rsa-dpub(bob-keym);
declare-public rsa-dpub(igor-keym);
declare-public rsa-dpriv(igor-keym);

procedure get-my-key [ID]
local ALICE, BOB;
```

```
{
  ALICE := alice; BOB := bob;
  if ID = ALICE
  {
    return rsa-dpriv(alice-keym);
  }
  if ID = BOB
  {
    return rsa-dpriv(bob-keym);
  }
  halt;
}

procedure get-her-key [ID]
local ALICE, BOB, IGOR;
{
  ALICE := alice; BOB := bob; IGOR := igor;
  if ID = ALICE
  {
    return rsa-dpub(alice-keym);
  }
  else
  {
    if ID = BOB
    {
      return rsa-dpub(bob-keym);
    }
    else
    {
      if ID = IGOR
      {
        return rsa-dpub(igor-keym);
      }
      else
      {
        halt;
      }
    }
  }
}

procedure ns-initiator [ID]
{
  IDENTITY := ID;

  receive PEER;
```

```
  get-my-key [IDENTITY] -> MYKEY;
  get-her-key [PEER] -> HERKEY;
  generate MYNONCE;
  MESSAGEA := rsa-enc(HERKEY, cons(MYNONCE, IDENTITY));
  send MESSAGEA;

  receive RESPONSE;
  DECRYPTED := rsa-enc(MYKEY, RESPONSE);
  FIRSTNONCE := car(DECRYPTED);
  if FIRSTNONCE = MYNONCE
  {
    HERNONCE := cdr(DECRYPTED);
    MESSAGEB := rsa-enc(HERKEY, HERNONCE);
    send MESSAGEB;
    halt-check not[[exists Z
                     [[not[Z.IDENTITY = ME.IDENTITY]]
                       and Z.HERNAME = alice
                       and Z.STATUS = good
                       and [not [ME.PEER = Z.IDENTITY]]]]];
  }
  else
  {
    halt;
  }
}

procedure ns-responder [ID]
{
  IDENTITY := ID;
  get-my-key [IDENTITY] -> MYKEY;
  receive INITMSG;
  DECRYPTED := rsa-enc(MYKEY, INITMSG);
  HERNONCE := car(DECRYPTED);
  HERNAME := cdr(DECRYPTED);
  ALICENAME := alice;

  get-her-key [HERNAME] -> HERKEY;
  generate MYNONCE;
  MESSAGE := rsa-enc(HERKEY, cons(HERNONCE, MYNONCE));
  send MESSAGE;
  receive RESPONSE;
  DECRYPTEDB := rsa-enc(MYKEY, RESPONSE);
  if DECRYPTEDB = MYNONCE
  {
    STATUS := good;
    halt;
```

```
  }
  else
  {
    STATUS := bad;
    halt;
  }
}

principal 1 ns-initiator[alice];
principal 1 ns-responder[bob];
```