Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Danmarks Tekniske Universitet
DTU Informatics

MASTER'S THESIS

Verifying security protocols by combining static analysis and model checking

by
Egbert Teeselink

Supervisors: H. Riis Nielson, A. Mathijssen,

Kongens Lyngby, 1-9-2008

ABSTRACT

In systems that require secure communication between agents, the protocols that are designed to make this as secure as required, often prove to be the weakest link in ensuring security. As a result, a lot of research has recently been done on methods to automatically verify certain properties about these protocols, so that we may find and correct flaws before a protocol is put to use.

LySa is a process calculus which has been specifically designed to specify the behaviour of security protocols. In contrast to many more common methods of specifying these protocols, LySa forces the author to be explicit about all details of the behaviour. The LySa Tool is a program that reads a protocol specified in the LySa calculus and uses static analysis techniques to verify certain properties.

Because the LySa Tool over-approximates the behaviour of the protocol in its analysis, no absolute certainty is provided if the tool reports a flaw. Additionally, the tool's output is often not enough to reconstruct an attack or to fix the flaw. By means of model checking, a common technology for verifying communicating systems, we hope to address these limitations. Using a toolset such as mCRL2, we gain absolute certainty about the behaviour of a protocol in a finite scenario and we can automatically construct an instance of an attack if a flaw is found.

We conclude that model checking using mCRL2 and LySa's static analysis can be combined into a strong and efficient means of verifying security protocols, and we describe how this can be done.

RESUMÉ

De protokoller der benyttes til sikker kommunikation i systemer, viser sig ofte at være det svageste led af sikkerheden. Dette har i den seneste tid resulteret i en del forskning i forskellige metoder til automatisk at verificere specifike egenskaber ved disse protokoller, så eventuelle fejl og mangler kan findes og vettes før en protokol tages i brug.

LySa er en proceskalkulen specielt designet til at specificere opførslen af sikkerheds-protokoller. Til forskel fra mange mere almindelige metoder til at specificere disse protokoller tvinger LySa udvikleren to at være eksplicit omkring alle detaljer af protokollens opførsel. "LySa Tool" er et program som læser en protokol specificeret i LySa kalkulen og benytte statisk analyse teknikker til at verificere protokollens egenskaber.

Fordi "LySa Tool" over-approksimerer opførslen af protokollen i dens analyse, er der ingen absolut sikkerhed hvis programmet reporterer et hul i en protokol. Desuden er programmets output ofte ikke nok til at rekonstruere et angreb eller rette fejler. Ved at benytte "model checking", en almindelig metode til at verificere kommunikerende systemer, håber vi på at kunne adressere disse begrænsninger. Ved at bruge et værktøjssæt så som mCRL2, opnår vi absolut sikkerhed omkring opførslen af en protokol i et endeligt scenarie og vi kan automatisk konstruere en instans af et angreb hvis et hul i protokollen bliver fundet.

Konklusionen er at ”model check” med mCRL2 og statisk analyse med LySa kan kombineres til en stærk og effektiv måde at verificere sikkerhedsprotokoller, og vi beskriver hvordan dette kan gøres.

SAMENVATTING

In systemen waarin beveiligde communicatie is gewenst zijn de protocollen die dit mogelijk maken vaak de zwakste schakel om de veiligheid ook echt te garanderen. Om deze reden is recentelijk er veel onderzoek gedaan naar methoden om bepaalde eigenschappen van deze protocollen automatisch te verifiëren, zodat fouten gevonden en gerepareerd kunnen worden voordat het protocol is gebruikt.

LySa is een procescalculus die speciaal is ontworpen om het gedrag van securityprotocollen in uit te drukken. In tegenstelling tot de gangbare methode om deze protocollen te specificeren, dwingt LySa de auteur om expliciet te zijn in elk detail van het gemodelleerde gedrag. De LySa Tool is een programma dat in LySa uitgedrukte protocollen kan lezen en met behulp van statische analyse eigenschappen van deze protocollen kan verifiëren.

Omdat de LySa Tool een overapproximatie maakt van het gedrag van het gegeven protocol, is er geen volledige zekerheid indien de tool een fout rapporteert dat deze ook echt bestaat. Ook is de uitvoer van de tool vaak te weinig specifiek om een aanval te reconstrueren of direct de fout te verhelpen. Doormiddel van model checking, een gangbare technologie voor het verifiëren van communicerende systemen, hopen wij deze beperkingen te omzeilen. Met behulp van een toolset zoals mCRL2 kunnen we absolute zekerheid verwerven omtrent het gedrag van een protocol in een eindig scenario. Indien een fout wordt gevonden, kunnen wij hiermee automatisch een aanval genereren.

De conclusie zal zijn dat model checking met mCRL2 en LySa's statische analyse gecombineerd kunnen worden tot een krachtige en effectieve manier om securityprotocollen te verifiëren.

PREFACE

This thesis is part of the work done for obtaining an M.Sc. degree. The work has been carried out at the DTU Informatics department of the Technical University of Denmark, under joint supervision of Hanne Riis Nielson of the Language Based Technology group of DTU and Aad Mathijssen of the Design and Analysis of Systems group of the department of Computer Science and Mathematics at Eindhoven University of Technology, the Netherlands. The project corresponds to 40 ECTS credits and ran from February 2008 to August 2008.

First of all I would like to thank Hanne Riis Nielson, Flemming Nielson, Aad Mathijssen and Jan Friso Groote for their feedback and encouragement throughout the project.

I would also like to thank the MSc. students, PhD. students, researchers and professors at the Language Based Technology group at DTU for their interest, support and companionship, and Aron Lindberg and Tania Surrow Larsen for helping me with the Danish translation of the abstract.

CONTENTS

Chapter 1 INTRODUCTION

As the world increasingly relies on electronic communication, secure communication over possibly insecure networks has become a commonplace requirement. Over the years, methods have been developed to allow systems to confidentially communicate, usually employing encryption to hide the contents of messages from anyone but the intended recipients. Protocols that have been designed to facilitate the initiation and continuation of such secure communication are called security protocols or cryptographic protocols. These protocols have proven to often be the weakest link in ensuring security, and vulnerabilities in them have sometimes been left undiscovered for many years.

As a reaction, various approaches have been developed that can be used for verifying security protocols. These are commonly divided into two distinct categories: first, there is theorem proving, where by hand or with help from the computer an attempt is made to formally prove that a certain protocol has certain specific features. Secondly, there is model checking, where usually all possible ways a protocol could behave are extracted into a transition system which can then be checked for certain properties. Model checking is fully automatic, but has the disadvantage that in general, only a limited number of scenarios can be represented, which means that certain attacks may not be found. Theorem proving does not have this problem, but is very labour intensive and requires a lot of work to be done for every protocol, every variant of it, and every property that needs to be verified. As such, fully automated solutions are often preferred.

With LySa [**2**], a third category has been added to this list. The LySa tool uses techniques from static source code analysis to detect possible flaws or vulnerabilities in protocols. The analysis is supported by a custom language, the LySa calculus, which allows one to very precisely describe the behaviour of a security protocol, as well as the properties it is supposed to meet. The static analysis of the LySa tool collects certain sets of information about a protocol specification without executing or simulating it. Instead, it performs an over-approximation, which means that any behaviour that *may* be exhibited by the protocol is included in the analysis. One advantage of this approach is that arbitrarily large systems may be analysed.

While this makes the technique an efficient and powerful approach, it has two important drawbacks. First of all, because an over-approximation is made, vulnerabilities may be reported that cannot, in fact, occur in the protocol. Results have shown that this only seldom occurs in practice, however it is still a reasonable drawback. Secondly, the LySa tool reports little information about how the actual flaw works; it can only tell us where in the protocol there may be a flaw, but not the steps by which a malicious agent may take advantage of it.

Such a list of steps, the *attack*, tends to clearly expose the weakness of a protocol and often makes fixing the flaw a simple task.

The goal of this project is to strengthen the LySa tool such that these two drawbacks may be remedied. By applying model checking techniques to protocols expressed in LySa, we are able to state with certainty whether or not attacks are possible, albeit only in finite scenarios. We take advantage of LySa's preciseness in specifying both the behaviour and the goals of a protocol. Furthermore, we may be able to automatically reconstruct the attack if a flaw is indeed found. Our approach is to automatically convert a LySa protocol to a specification in the mCRL2 process algebra. In doing this, we gain access to a powerful set of tools and formalisms that allow us to perform a wide range of analyses. As a side-effect, our result shows that indeed the mCRL2 language and toolset are well suited for the analysis of security protocols.

1.1 OVERVIEW OF THE THESIS

In this chapter, we introduce the problem and present a rough outline of our solution. Additionally, the general setting of network security will be introduced, as well as a running example that we will refer to throughout the thesis. In Chapter 3, we will propose Typed LySa, a process calculus that can be used to express the behaviour of security protocols clearly and unambiguously. Typed LySa is the language that we express all protocols that we wish to verify in. It is only a minor variation of LySa [**2**] and the differences between the two will be discussed in section 3.7.

The mCRL2 process algebra [**14**] is introduced in Chapter 4. mCRL2 is a process algebra suited for expressing a wide variety of systems and protocols, including security protocols as we will see. mCRL2 comes with an extensive toolset that can be used for automatically verifying properties of these systems through model checking. In Chapter 5, we will show how a protocol expressed in Typed LySa can be converted to an mCRL2 process so that the mCRL2 toolset may be used for verifying that the protocol is secure or for finding an attack if it is not.

Finally, in Chapter 6, we present some results of well-known security protocols and the attacks found and we will conclude that model checking is an effective complement to static analysis of security protocols.

Chapter 2 THE PROBLEM AND THE SOLUTION

2.1 THE NETWORK SECURITY CONTEXT

In open networks such as, for example, the internet, there often is a need for parties to communicate securely. Applications such as internet banking and confidential e-mailing are obvious examples, but many more exist. Also, with the world's increasing dependence on electronic systems, scenarios where communication that takes place on closed networks is read or modified, be it by outsiders or by malicious or careless employees, are very realistic.

2.1.1 SECURITY ON INSECURE NETWORKS

The need for secure communication has been addressed by the field of cryptography, which has produced numerous methods for encrypting information in ways such that only selected parties may be able to decrypt and use sent information. Cryptography alone, however, is not enough to warrant secure communication. The central idea in cryptography is that some *key* is used to mangle a piece of data in such a way that only when in possession of a decryption key, the data may be restored to original form. Without exactly the correct decryption key, useless garble will be found instead. Another problem entirely, however, is how to make sure that the right parties obtain the right keys and any other parties don't. This problem is called *key establishment* and is one of the two common goals of security protocols. Obviously, if a malicious attacker can somehow obtain a correct decryption key, security will be compromised no matter how strong the encryption scheme is.

The second common goal of security protocols is *authentication*, which informally means that parties obtain certainty that they are indeed communicating with the intended party. Both key establishment and authentication have many different definitions throughout the literature, which we will not discuss because our formalism for specifying security protocols, Typed LySa, provides a very precise means for exactly specifying the goals of a protocol. For an in-depth discussion of various security protocol goals, please see [**3**] and [**16**].

Security protocols are generally described as a sequence of messages that parties may send to one another, after which one or more goals (hopefully) have been reached. Typically, after a key establishment protocol has successfully been run, two parties (commonly called *A* and *B*), both know a new key *K* that they can subsequently use to exchange information securely with. Often, a trusted server *S* is involved in a protocol, with which both *A* and *B* are assumed to already share a *master key*. How this master key was established is left unspecified.

VERIFYING SECURITY PROTOCOLS

Even though security protocols are often short and simple protocols, often consisting of no more than 5 messages, verifying their correctness is not a simple task. Many protocols that

have been designed were later shown to be flawed, some no less than 17 years after their conception [**17**]. The problem haunting security protocols is that they are deployed in insecure networks. In such networks, malicious outsiders may be able to read or even modify network traffic, at any moment, in uncountable ways. As such, we cannot simply study a security protocol in isolation: we must assume that a protocol runs in the most dangerous and uncertain environment that is still realistic.

2.1.2 THE DOLEV-YAO ATTACKER

Dolev and Yao [**10**] proposed a set of assumptions that define a realistic, yet worst-case environment in which security protocols may run. These assumptions are, freely after Boyd and Mathuria [**3**]:

1. The attacker is able to eavesdrop on all messages sent in a cryptographic protocol.
2. The attacker is able to alter, block and re-route all messages sent in a cryptographic protocol using any information available. In addition, the attacker can generate and insert completely new messages.
3. The attacker may be a legitimate protocol participant (an insider) or an external party (an outsider) or a combination of both.
4. The attacker may start any number of parallel protocol runs between any agents, including different runs involving the same agents and with agents taking the same or different protocol roles.

In addition to the Dolev-Yao assumptions, we often make a fifth assumption [**3**] if we want to ensure that a protocol is secure even after many runs:

5. An attacker is able to obtain the value of a session key used in any sufficiently old previous run of the protocol.

Obviously, when combined with an attacker following these five assumptions, showing that a protocol indeed accomplishes its goals is a complex matter.

An attacker that follows the first four assumptions and takes advantage of them as much as possible is generally called a *Dolev-Yao attacker*. Naturally, it is hardly realistic that a process performing all of a Dolev-Yao attacker's actions is active on a physical network, but if we can show that a protocol can withstand a Dolev-Yao attacker, then we are reasonably certain that any real-world attacks will fail as well, even if the attacker has full control over the network.

2.1.3 AN EXAMPLE

As a running example in this work we use a variation of the well-known Wide Mouthed Frog protocol [**6**]. Its goal is to establish a new key between two agents A and B, who are both assumed to have a master key with a mutually trusted server S. A simple (but flawed) version of this protocol is taken from [**11**]. In the traditional notation for protocol narrations, the protocol runs as follows:

1. $A \rightarrow S: A, \{B, K\}_{K_{AS}}$
2. $S \rightarrow B: \{A, K\}_{K_{BS}}$
3. $A \rightarrow B: \{m\}_K$

In this notation, "$X \rightarrow Y: M$" means that some agent X sends a message M to some agent Y. Furthermore, $\{x_0, \dots, x_k\}_y$ means that some list of values $x_0, \dots, x_k$ is encrypted using the key y.

In this particular protocol, A is the initiating agent which is assumed to have a master key K_{AS} that it shares with the server. In message 1, A generates a new key K which it intends to share with agent B and sends those values to the server, encrypted using the key K_{AS} so that the server can decrypt it. Additionally, A includes its own name in clear text to serve as a hint to the server about which key to use for decrypting $\{B, K\}_{K_{AS}}$. Server S receives this message, decrypts it, and extracts the key K. In message 2, S sends this key together with A's name to B, encrypted by the key K_{BS} which B and S are assumed to share. Finally, A can use this key to communicate privately with B, for instance by sending some value m in message 3. The whole structure is shown visually in Figure 2.1.

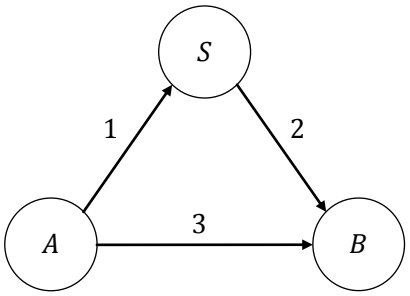

Figure 2.1. Structure of the Wide Mouthed Frog protocol

The Wide Mouthed Frog protocol is an interesting study object because small changes to it can significantly affect its security properties. Additionally, the protocol is short and simple, which makes it ideal as a running example because transformation of the protocol to different forms and languages will still be relatively short and simple.

AN ATTACK

The aforementioned protocol exhibits a serious flaw, related to key freshness. Imagine that during some run of the protocol an attacker I who is able to eavesdrop on all network communication intercepts the second message in the protocol, $\{A, K\}_{K_{BS}}$. He can replay this message to B at will, and B will believe that it was in fact S sending the message. Now, recall the 5th assumption in section 2.1.2 which states that it may be possible for an agent to obtain the value of any sufficiently old key. This means that after a long enough period of time, the attacker may obtain the value of key K. When B then receives $\{A, K\}_{K_{BS}}$, it thinks that this is a new message for communication with A, even though A was never involved. The attacker and B will now engage in secure communication using key K and B may tell the attacker secrets that were intended for A only. Schematically, the attack can be narrated as follows:

1. $A \rightarrow S: A, \{B, K\}_{K_{AS}}$
2. $S \rightarrow I(B): \{A, K\}_{K_{BS}}$

   …
3. $I(S) \rightarrow B: \{A, K\}_{K_{BS}}$
4. $I(A) \rightarrow B: \{m\}_K$

Here, *I* is the attacker, with the role it assumes in parentheses.

2.1.4 PROTOCOL NARRATIONS

Security protocols are commonly expressed in a notation that is very easy to read and understand but also has a few shortcomings. This notation simply consists of a list of messages as they are sent from one party to another; we've seen an example in section 2.1.3.

There are several problems with this notation. First of all, we can only show which messages are sent in a successful run, but we cannot tell what defines a successful run or what happens if wrong information is sent. More specifically, we cannot precisely tell which checks an agent must perform when a message is received before continuing the protocol. Common sense tells us that, for instance, when an agent receives a value that it expects to be equal to one that it sent earlier, it should check that these two values are indeed equal. However, such conclusions are left entirely implicit.

Secondly, the traditional notation provides us with no way to express the variety of scenarios a protocol may be run in. More exactly, *roles* and *agents* cannot be distinguished from one another. In traditional notation, we usually have some agent A, some agent B and maybe some server S, and they communicate. In practise, there may be many agents playing roles A and B, agents may perform both roles and many protocol runs could be executed simultaneously. Additionally, initial information such as master keys shared between an agent and a server, may be related or unrelated to the role an agent plays.

Finally, from a traditional protocol narration, it is impossible to deduct which security properties it is supposed to fulfil. Commentaries by the authors of classic security protocols often remain vague about this as well, for instance by specifying only that a protocol is intended to authenticate two parties to one another. As shown in [**16**], for example, there exist many different interpretations for the term *authentication* alone. As a resort, many commonly required security properties have been formalised, but even these are not always directly applicable to a protocol without making additional assumptions.

2.2 THE BIG PICTURE

In this work we attempt to combine the powers of the LySa tool and the mCRL2 toolset. In this section, we show why this makes sense.

2.2.1 LYSA AND THE LYSA TOOL

LySa is a process calculus designed for specifying the behaviour of security protocols. These protocols are generally concerned with the establishment or transport of *keys* that are to be used for private communication on an insecure network. The central question is whether a certain protocol is secure enough for its use. This means that we wish to find out if a protocol ensures certain security properties, even when run in an insecure environment. LySa allows us to specify both the behaviour of the protocol and the properties the protocol is expected to have. The latter can be done by adding *annotations* to the protocol code.

The LySa tool uses static analysis to verify whether these properties hold in a protocol. The analysis technique is static, which means that the protocol is not executed or simulated, but an attempt is made to derive some useful information by simply inspecting the source code and relating parts of it to one another. The LySa tool can do this by making a strict *over-approximation* of the behaviour of the protocol. This means that some behaviour may be

reported by the tool's analysis that can in fact never take place, were the protocol executed. However, the opposite is never true. This means that if the LySa tool reports no flaws, we are certain that there can indeed be none (given LySa's general assumptions about the environment). It is important to make a clear distinction between the LySa *calculus* and the LySa tool's *analysis*. Figure 2.2 illustrates this relationship.

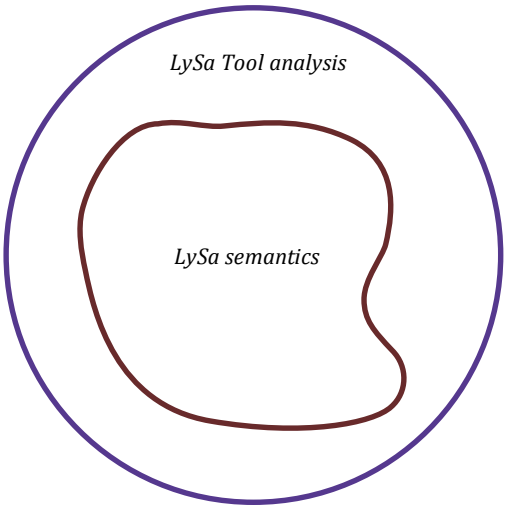

Figure 2.2. The behaviour of a protocol as specified in LySa vs. the behaviour seen by the LySa Tool

The inner shape represents the behaviour of a protocol as specified in LySa. This behaviour could be expressed, for example, in terms of a labelled transition system. The LySa tool, however, derives conclusions from what would be a strictly larger transition system, if it were ever constructed. This is shown by the outer circle. It is guaranteed that all behaviour that the protocol exhibits according to LySa's semantics is also seen by the LySa tool's analysis.

2.2.2 Typed LySa

Typed LySa is a variant of the LySa calculus where the types of newly read and decrypted values are explicitly specified. It was designed primarily to ease the transition to mCRL2 but is a useful language for describing a protocol's behaviour in its own right. When specifying a protocol in Typed LySa, we assume that agents can see the structure of a message sent: it can uniquely identify each element in a message, and determine whether it is an atomic value or a *ciphertext* that, given the correct key, may be decrypted and split up into elements in the same way.

All behaviour that can be specified in Typed LySa can also be specified in LySa, but in LySa, more behaviour can be found. In other words, when we remove all type specifiers from a Typed LySa protocol and simulate it with LySa's semantics, we are making an over-approximation of the Typed LySa protocol. This is shown in Figure 2.3.

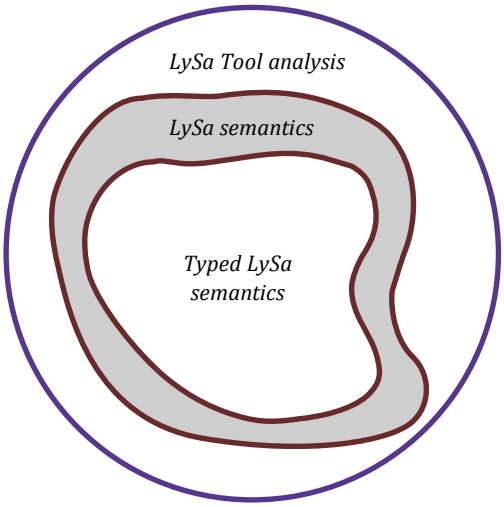

Figure 2.3. The behaviour of a protocol in the LySa tool, in LySa, and in Typed LySa

If, in the implementation of a protocol, indeed the assumption holds that atomic values and ciphertexts can be distinguished, then whatever is in the grey area may be safely disregarded.

2.2.3 mCRL2 and the mCRL2 toolset

We can transform a Typed LySa protocol narration into an mCRL2 process. mCRL2 is a language for modelling communicating systems and a toolset for verifying these. We can model check this process by exploring its complete state space and checking whether the required properties are never broken. While it is possible to express the presence of infinitely many communicating agents in both (Typed) LySa and mCRL2, it is obvious that we cannot traverse an infinitely sized state space and terminate. As such, when transforming a Typed LySa specification into mCRL2, we apply some artificial limit to this infinite size, for instance, we can simulate only the presence of a finite number of agents. This means that by transforming a Typed LySa specification into mCRL2 and exploring its state space, we in fact *under-approximate* the behaviour described in Typed LySa, completing our diagram in Figure 2.4.

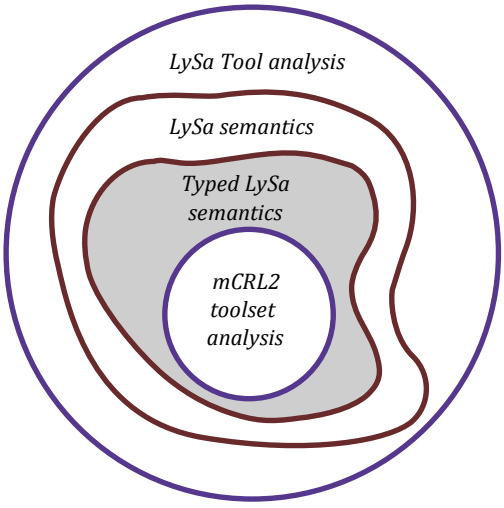

Figure 2.4. The behaviour of a protocol in the LySa tool, in LySa, in Typed LySa and in mCRL2

If the protocol narration contains no infinite constructs, the grey area is empty, and the mCRL2-generated state space exactly corresponds to the specified behaviour of our protocol.

2.2.4 EVERYTHING COMBINED

The application of this combination of tools now becomes clear: Typed LySa is a very precise and explicit language for specifying the behaviour of security protocols, in comparison to more traditional methods of narrating these protocols. We can then first use the (fast) LySa tool to quickly check whether there is a chance for flaws at all. If LySa reports a flaw, we use mCRL2 to inspect the state space and obtain certainty, along with the necessary information for reconstructing (and fixing) the flaw.

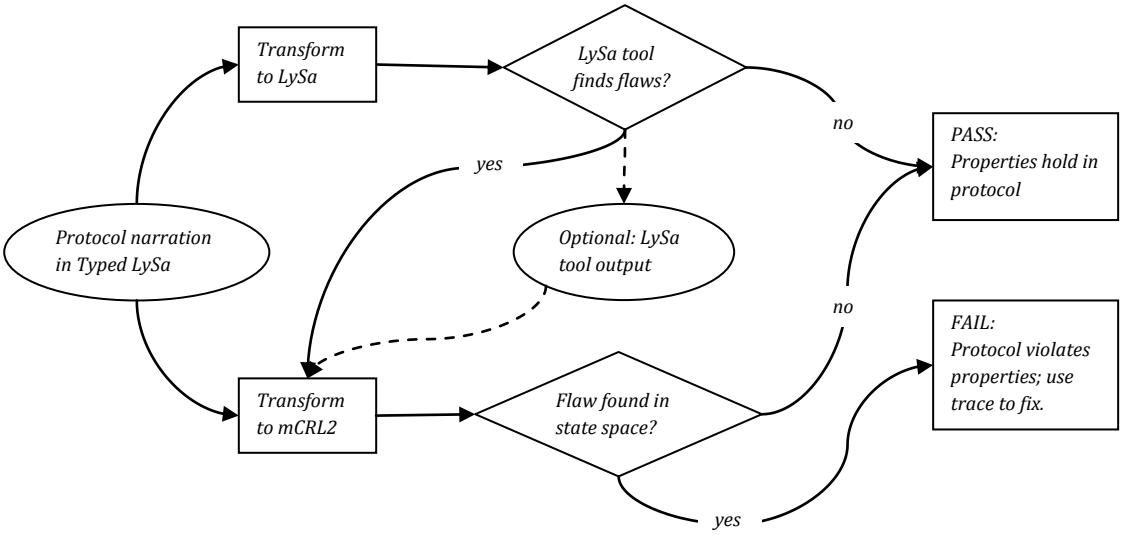

Figure 2.5: Flow chart showing how various tools and calculi may be combined

The whole scenario is outlined in Figure 2.5. As shown in the diagram, an obvious combination of the LySa tool and the mCRL2 toolset analysis is to somehow use the output of the LySa tool to optimise the size of the state space generated with mCRL2. In this work this option has not been explored in detail, however. Nevertheless, it may be interesting in future research.

The central novelty in this work is the automated conversion from Typed LySa to mCRL2; all other steps in Figure 2.5 can be done using already existing tools. The formal conversion rules by which such a transformation may be performed are detailed in Chapter 5. A command-line tool that implements these rules is described in Appendix A.

Figure 2.5 suggests that the whole process can be automated, and this certainly is the case. Nevertheless, to allow the user direct control over what is done at which point, we do not provide such an “all-encompassing” tool that makes calls to the various tools used.

Chapter 3 TYPED LYSA

Process calculi have been used for decades to describe and study the behaviour of communicating systems and protocols. Famous examples include Milner's π-calculus[**19**] and Hoare's CSP [**15**]. Additionally, calculi for specific problems have been designed, such as the Spi-calculus [**11**], a variation of the π-calculus specifically intended for describing security protocols.

In this chapter we propose Typed LySa, a process calculus for clearly and unambiguously expressing the behaviour of security protocols. Typed LySa is in many ways a simplification of the Spi-calculus and is intended to solve the three problems discussed in section 2.1.4. This is generally done by forcing the protocol designer to be specific about every detail of the protocol's execution. The behaviour and configuration of agents in a protocol as well as the exact security properties that this protocol is believed to have, are explicitly expressed in a Typed LySa protocol specification.

Typed LySa is merely a small variation of the LySa calculus as proposed in [**2**,**4**]. The differences between the two calculi are discussed and accounted for in section 3.7. In the first three sections, we will merely focus on the *object-level* of Typed LySa, with which it is possible to describe single protocol runs. In section 3.4 we will add a *meta-level* which allows us to specify more complex situations. The object-level and meta-level together define the complete Typed LySa calculus.

3.1 THE SYNTAX OF TYPED LYSA

Typed LySa is a small and straightforward calculus. It has constructs for parallel processes, sequential execution, for sending and receiving messages and for encrypting and decrypting messages. Any Typed LySa expression is either a data expression or a process expression. Before we provide a grammar and formal semantics, we will informally discuss the meaning of each Typed LySa expression.

Given the set of names *Name* and the set of variables *Var*, Typed LySa allows data expressions E as listed in Table 3.1:

$n \in Name$	A name: an explicit unique value that can for example be used as a nonce, a key or the address of an agent. It is the elementary building block of any Typed LySa data expression.
$x \in Var$	A variable
$\{E_1, \dots, E_k\}_{E_0}$	A ciphertext: a list of data expressions $E_1, \dots, E_k$ encrypted with key E_0, which can also be any data expression.

Table 3.1 Data expressions in Typed LySa

Furthermore, Typed LySa allows process expressions P as listed in Table 3.2:

0	The empty process.
$P_1 \mid P_2$	The parallel operator, which runs the two processes P_1 and P_2 in parallel.
$(\nu\ n)\ P$	The restriction operator, which generates a new name n that is restricted to the scope of P.
$\langle E_0, \dots, E_{n-1} \rangle . P$	Send a message consisting of the expressions $E_0, \dots, E_{n-1}$ and sequentially execute P.
$(E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}) . P$	Receive a message with pattern matching. This means that the message will be accepted if and only if it consists of n values, $E_0, \dots, E_{k-1}$ correspond exactly to the first k values of a sent message and the types $T_k \dots T_{n-1}$ correspond to the types of the last $n - k$ values of a sent message. If this pattern match succeeds, then P is executed with the variables $x_k, \dots, x_{n-1}$ bound to the corresponding received values.
`decrypt` E `as` $\{E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{E_n}$ `in` P	Decrypt a ciphertext E with pattern matching, which means that it succeeds if and only if E is a ciphertext of length n, where E is encrypted with key E_n, where the first k values correspond to expressions $E_0, \dots, E_{k-1}$ and the types of the last $n - k$ values correspond to the types $T_k \dots T_{n-1}$. If the pattern match succeeds, then P is executed with the variables $x_k, \dots, x_{n-1}$ bound the corresponding decrypted values.

Table 3.2 Typed LySa process expressions

It is important to note that Typed LySa has no notion of channels: all communication takes place on one global network, the *ether*. From a security perspective, this corresponds to most real-life communication networks, where malicious attackers can potentially read and change everything that is being sent.

If it is necessary that communicating agents are be able to distinguish the origin and destination of a message, the addresses of these agents can be made part of the message, thus simulating an insecure channel. This corresponds to how for instance the IP protocol works, where the origin and destination addresses of a packet are sent unencrypted. For example, an agent A that wishes to send a message to B containing some value m is typically expressed as

$$\langle A, B, m \rangle$$

The receiving partner B can then use pattern matching in the input operator to read only such a message:

$$(A, B; xm: N)$$

Where the value m sent by agent A will be bound to the variable xm in agent B's scope. A more elaborate example will be given in section 3.1.2.

Another important characteristic of Typed LySa is the fact that there is no support for many common data types or operations. Unlike its name might suggest, Typed LySa does not specifically distinguish keys, nonces, addresses, etcetera. The only data types are the *name* and the *ciphertext*, and the amount of names present in any object-level process is finite. This also means that the variable types T_i in the process expressions for receiving and decrypting can only be N or C, for *name* and *ciphertext*, respectively.

The ν operator can be used to restrict the scope of a name to a specific part of a Typed LySa process. We say that some process expression P *knows* some name α if this name exists in P's scope. In other words, $(\nu\ \alpha)\ P$ means that we wish that P (and only P) knows α. Finally, one of the well-formedness restrictions of Typed LySa dictates that all names must be restricted before they are used, see section 3.6.2.

3.1.1 Grammar

Presently we propose a grammar for Typed LySa. Note that this grammar describes only that part of the syntax of Typed LySa that has been introduced so far, that is, it includes only object-level elements and no *annotations*. For a complete grammar, please see section 3.6.1.

$$
\begin{aligned}
E &::= a \mid x \mid \{E_0, \dots, E_{n-1}\}_{E_n} \\
T &::= N \mid C \\
P &::= \langle E_0, \dots, E_{n-1} \rangle . P \\
&\mid (E_0, \dots, E_{k-1}; x_k : T_k, \dots x_{n-1} : T_{n-1}) . P \\
&\mid \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1};\ x_k : T_k, \dots x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P \\
&\mid (\nu\ a)\ P \\
&\mid P_1 \mid P_2 \\
&\mid 0
\end{aligned}
$$

Either $(0 \leq k < n)$ *or* $(0 < k \leq n)$ *must hold in each rule*

Table 3.3. The grammar of object-level Typed LySa

In Table 3.3, an x represents a variable and an a represents a name. All other identifiers are constants or defined in the grammar.

3.1.2 An example

We can now express the Wide Mouthed Frog protocol as introduced in section 2.1.3 in Typed LySa. In this section, we model only a single run of the protocol, which we will extend throughout the rest of this chapter. The first observation we make is that there are three agents, A, B and S, which operate independently from one another. This means that our Typed LySa process will have the following global structure:

$$P_A \mid P_B \mid P_S$$

The second observation is that in Typed LySa, sending a message, receiving a message and decrypting a ciphertext are three separate actions. This means that each step in the original protocol narration has to be split into at least 2 actions; one for sending, one for receiving,

plus a decrypt action for every received value that should be decrypted. Recall that this protocol is specified in traditional narration format as follows:

1. $A \rightarrow S: A, \{B, K\}_{K_{AS}}$
2. $S \rightarrow B: \{A, K\}_{K_{BS}}$
3. $A \rightarrow B: \{m\}_K$

We shall now transform the messages step by step, and we start with message 1:

$$
\begin{array}{l}
(\nu\ A)(\nu\ B)(\nu\ S)(\nu\ K_{AS})(\nu\ K_{BS}) \\
(\\
\quad (\nu\ K) \\
\quad \langle A, S, A, \{B, K\}_{K_{AS}} \rangle. \\
\quad \ldots \\
| \\
\quad \ldots \\
| \\
\quad (A, S, A; z1: C). \\
\quad \texttt{decrypt}\ z1\ \texttt{as}\ \{B; zK: N\}_{K_{AS}}\ \texttt{in} \\
\quad \ldots \\
)
\end{array}
$$

On the first line, we declare the names that are to be known by all parties. These are the addresses of agents A, B and S, plus the master keys K_{AS} and K_{BS} that agents A and B share with the server. [1]

As we can see, agent A first generates a new key K. Subsequently, A prepends his own address and that of the server, S, to message 1 and sends it. S first receives the plain message and ensures that it is indeed a message from A intended for S using pattern matching. The fourth value in the message must be a ciphertext and is bound to the new variable $z1$. Subsequently, S attempts to decrypt the value in $z1$, once more applying pattern matching to ensure that the key contained is intended for communication with agent B.

Step 2 can now be added as follows:

[1] Note that it is not possible to express precisely that A and S, but not B, know key K_{AS} if we also want to model the dual case for K_{BS}. This is no problem in practice, however, because we, the modellers of the protocol, are in control of which names are used by which (honest) agents.

$$
\begin{array}{l}
(\nu\ A)(\nu\ B)(\nu\ S)(\nu\ K_{AS})(\nu\ K_{BS}) \\
(\\
\quad (\nu\ K) \\
\quad \langle A, S, A, \{B, K\}_{K_{AS}} \rangle. \\
\quad \ldots \\
| \\
\quad (S, B; y1 : C). \\
\quad \texttt{decrypt}\ y1\ \texttt{as}\ \{A; yK : N\}_{K_{BS}}\ \texttt{in} \\
\quad \ldots \\
| \\
\quad (A, S, A; z1 : C). \\
\quad \texttt{decrypt}\ z1\ \texttt{as}\ \{B; zK : N\}_{K_{AS}}\ \texttt{in} \\
\quad \langle S, B, \{A, zK\}_{K_{BS}} \rangle. \\
\quad 0 \\
)
\end{array}
$$

Notice that the server sends to B whatever value that zK is bound to; it cannot be certain whether it is really the key that A generated, only that it is the value that it received. We end the server process with 0, the empty process, because after step 2 it is done.

We can now finalise the Typed LySa narration with step 3:

$$
\begin{array}{l}
(\nu\ A)(\nu\ B)(\nu\ S)(\nu\ K_{AS})(\nu\ K_{BS}) \\
(\\
\quad (\nu\ K) \\
\quad \langle A, S, A, \{B, K\}_{K_{AS}} \rangle. \\
\quad (\nu\ m) \\
\quad \langle A, B, \{m\}_{K} \rangle. \\
\quad 0 \\
| \\
\quad (S, B; y1 : C). \\
\quad \texttt{decrypt}\ y1\ \texttt{as}\ \{A; yK : N\}_{K_{BS}}\ \texttt{in} \\
\quad (A, B; y2 : C). \\
\quad \texttt{decrypt}\ y2\ \texttt{as}\ \{; m : N\}_{yK}\ \texttt{in} \\
\quad 0 \\
| \\
\quad (A, S, A; z1 : C). \\
\quad \texttt{decrypt}\ z1\ \texttt{as}\ \{B; zK : N\}_{K_{AS}}\ \texttt{in} \\
\quad \langle S, B, \{A, zK\}_{K_{BS}} \rangle. \\
\quad 0 \\
)
\end{array}
$$

Now we are done expressing the behaviour of the Wide Mouthed Frog protocol in Typed LySa. In order to use this specification for verifying the protocol in a realistic setting, however, we typically wish to include more information. For example, we will want to explicitly embed an attacker, specify the scenario in which multiple agents act out multiple roles, and precisely define the security properties that the protocol is supposed to exhibit. We will show how this is done by gradually expanding the example in the following sections.

3.2 THE SEMANTICS OF TYPED LYSA

In order to be able to prove properties about Typed LySa and its conversion to mCRL2, we need to formalise the semantics of Typed LySa. Because Typed LySa closely resembles LySa, we base its semantics on the LySa semantics as given in [**4**]. Additionally, we take some ideas from LySa^{NS}, an extended version of LySa described in [**5**].

In this section, we will only provide the semantics for the object-level of Typed LySa, which means we disregard the meta-level in which scenarios can be specified. The semantics of the meta-level will be given in section 3.4. Additionally, the version of Typed LySa presented in this thesis does not support asymmetric encryption. Otherwise, the rules in this chapter are very similar to section 2.2.2. of Buchholtz' PhD thesis [**4**].

3.2.1 STRUCTURAL CONGRUENCE

The semantics of LySa and Typed LySa are defined along the concepts of Structural Operational Semantics [**21**]. In order to easily provide a *reduction semantics* of Typed LySa without stumbling over syntactical details, we first define a structural congruence relation ≡. This relation defines that two processes are considered equal if they only differ syntactically, in a way that substituting one by another does not change the way the process may execute.

$$P \equiv P$$

$$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$$

$$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$$

$$P_1 \equiv P_2 \Rightarrow \begin{cases} \langle E_0, \dots, E_{n-1} \rangle.\, P_1 \equiv \langle E_0, \dots, E_{n-1} \rangle.\, P_2 \\ (E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}).\, P_1 \equiv (E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}).\, P_2 \\ \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P_1 \equiv \\ \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P_2 \\ (\nu\ n)P_1 \equiv (\nu\ n)P_2 \\ P_1 | P_3 \equiv P_2 | P_3 \end{cases}$$

$$P_1 | P_2 \equiv P_2 | P_1$$

$$(P_1 | P_2) | P_3 \equiv P_1 | (P_2 | P_3)$$

$$P | 0 \equiv P$$

$$(\nu\ n)0 \equiv 0$$

$$(\nu\ n_1)(\nu\ n_2)P \equiv (\nu\ n_2)(\nu\ n_1)P$$

$$(\nu n)(P_1 | P_2) \equiv P_1 | (\nu\ n)P_2 \qquad \text{if } n \notin \text{free names}(P_1)$$

$$P_1 \overset{\alpha}{\Leftrightarrow} P_2 \Rightarrow P_1 \equiv P_2$$

Table 3.4. Structural congruence for Typed LySa

Table 3.4 is nearly identical to the one in [**4**]; types are added to the input and decryption operations and asymmetric key support is left out. Additionally, Typed LySa does not have a separate replication operator like LySa does; this will be discussed in more detail in section 3.4.

The free names in a process P are defined by those names that occur in any data expression in P but are not previously restricted (bound) by a ν expression. Similarly, free variables are variables that occur without having been previously bound by an input or decrypt operation. It should be noted here that syntactically, it is not possible to distinguish a free name from a free variable, as it is a plain, undeclared, identifier.

The last rule in Table 3.4 states that if P_1 can be alpha-renamed to P_2 (and vice versa), then they are equivalent. By α-renaming a process we mean that *bound* names and variables may

be replaced by different identifiers. This rule is important because in this work semantics are often defined by making substitutions. We invariably assume that if a substitution in process P causes a previously free name or variable to be indistinguishable from one that is bound in P, the bound name or variable is silently α-renamed to a different, unused identifier.

3.2.2 Reduction relation

Finally, we are ready to define the actual semantics of Typed LySa with a reduction relation in Table 3.5. This relation is also very similar to the one proposed by Buchholtz for LySa [**4**], except that we also leave out asymmetric encryption here. The values D_i are meant to be any Typed LySa data expressions that do not contain variables. In other words, they are either literal names or literal ciphertexts.

[Comm]	$\langle D_0, \dots, D_{n-1}\rangle.\, P_1 \mid (D_0, \dots, D_{k-1}; x_k{:}\,T_k, \dots, x_{n-1}{:}\,T_{n-1}).\, P_2 \;\rightarrow\; P_1 \mid P_2\left[x_k \overset{\alpha}{\rightarrow} D_k, \dots, x_{n-1} \overset{\alpha}{\rightarrow} D_{n-1}\right]$ $\left(\forall_{k \le i < n-1} T_i = \text{typeof}(D_i)\right)$
[Decr]	$\texttt{decrypt}\ \{D_0, \dots, D_{n-1}\}_{D_n}\ \texttt{as}\ \{D_0, \dots, D_{k-1}; x_k{:}\,T_k, \dots, x_{n-1}{:}\,T_{n-1}\}_{D_n}\ \texttt{in}\ P \rightarrow P\left[x_k \overset{\alpha}{\rightarrow} D_k, \dots, x_{n-1} \overset{\alpha}{\rightarrow} D_{n-1}\right]$ $\left(\forall_{k \le i < n-1} T_i = \text{typeof}(D_i)\right)$
[Onew]	$\dfrac{P \rightarrow P'}{(\nu\ n)\ P \rightarrow (\nu\ n)\ P'}$
[OPar]	$\dfrac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2}$
[Congr]	$\dfrac{P \equiv P' \quad P' \rightarrow P'' \quad P'' \equiv P'''}{P \rightarrow P'''}$

Table 3.5. Reduction relation → for Typed LySa

Additionally, we define the typeof operation over Typed LySa data expressions as follows, where a is a name and E_i is any data expression:

$$\text{typeof}(a) = N$$
$$\text{typeof}\left(\{E_0, \dots, E_{n-1}\}_{E_n}\right) = C$$

In the first two rules we perform capture-avoiding substitution, which we write $P[x \overset{\alpha}{\rightarrow} a]$. By this we mean that if P has locally bound names that are also called a, then these will be silently α-renamed to different (unused) names before every x is substituted by a.

The first rule, [Comm], enables synchronous communication and formalises the semantics of the input and output operators as described in section 3.1. For communication to succeed, we require that the first k values being sent are equal to the first k values being received. Furthermore, the total number of values in the message, n, must be equal in both operations. Finally, we require that the types of the last $n - k$ values sent correspond to the types we expect. For example, a communication cannot occur if a value that is sent is a ciphertext when a name was expected, or vice versa. We call these three requirements the *pattern-matching requirement*; the sent message has to match the specified pattern for communication to occur.

If the communication can occur, all variables $x_k \dots x_{n-1}$ are substituted (with alpha-renaming) by the communicated values $D_k \dots D_{n-1}$ in P_2, the remainder of the right-hand process. Notice that this way it is ensured that only literal names and ciphertexts occur in the parts of a

process that could enable a reduction step to be performed; the moment a variable becomes available for use in a process, it is immediately substituted by the value it would be bound to.

For decryption, we assume that an agent can decrypt a value if and only if it knows the correct key. Additionally, we assume that messages contain enough redundancy for an agent to verify that indeed the correct key was used. In other words, an agent can determine whether or not a key "fits" a certain ciphertext. This assumption is formalised in [Decr]; decryption can occur if and only if the same key D_n is used in the ciphertext being decrypted and in the pattern matching expression. Other than that, [Decr] has essentially the same semantics as [Comm]: a similar pattern matching requirement is enforced, after which all new variables get substituted by the newly decrypted values.

[Onew] simply states that any restricted names in some process P must still be restricted in the process that it may reduce to. [Opar] allows processes in parallel to do steps independently from one another, and finally [Congr] states that we may use any of the rules in Table 3.4 to transform a process P into a shape P' if that allows P' to do a reduction step.

3.3 ANNOTATIONS

As introduced in section 2.1.4, it is important to precisely formulate the properties a protocol should exhibit before one makes an attempt to verify that such properties hold. In Typed LySa (as in LySa), this can be done by embedding *annotations* inside the protocol description. These annotations are additional pieces of code that do not have any semantics in the behaviour of the protocol; their content does not influence the behaviour of the protocol. Instead, we use annotations to describe what we wish or expect the protocol to guarantee. By embedding annotations directly in the protocol specification, rather than separately specifying certain common security goals such as *key confidentiality* or *aliveness*, we have very precise control of what property should hold at which point in time and leave no room for ambiguity.

3.3.1 DESTINATION/ORIGIN AUTHENTICATION

Security protocols typically seek to establish some form of authentication between agents. The term "authentication" is subject to many different interpretations [**16**], so we must be more specific about exactly what we mean. In Typed LySa annotations, we specify instances of what is called *destination/origin authentication* [**2**]. Unlike most authentication formalisations, destination/origin authentication is not used to express a property of the whole protocol, but of a particular piece of data.

More specifically, when a list of values are encrypted into a ciphertext, we specify at which points it may be decrypted. Conversely, upon decryption of a ciphertext, we specify at which points it may have been encrypted. For example, consider the third step in our running example, the Wide Mouthed Frog protocol:

$$A \rightarrow B: \{m\}_K$$

In isolation, we can express this message in Typed LySa as follows:

$$
\begin{array}{l}
(\nu\ K) \\
(\\
\quad (\nu\ m)\ \ \langle A, B, \{m\}_K \rangle.\, 0 \\
| \\
\quad (A, B; y\colon C). \\
\quad \texttt{decrypt}\ y\ \texttt{as}\ \{; ym\colon N\}_K\ \texttt{in}\ 0 \\
)
\end{array}
$$

Now, with destination/origin authentication, we can express that the ciphertext $\{m\}_K$ created by agent A may only be decrypted by agent B. Moreover, we can be as precise as specifying that agent B may only decrypt the ciphertext at a certain point in the protocol, and that agent B may only decrypt a value if it was encrypted by agent A, at a specific point.

We do this by giving names to points in the protocol where encryption and decryption occur. These names are called *cryptopoints* and can be chosen at will. Then, upon encryption, we include a list of all cryptopoints where we allow the created ciphertext to be decrypted: the set of destination cryptopoints. Similarly, upon decryption, we include the set of origin cryptopoints where the ciphertext may have been created. As such, we annotate our example as follows:

$$
\begin{array}{l}
(\nu\ K) \\
(\\
\quad (\nu\ m)\ \ \langle A, B, \{m\}_K\ [\texttt{at}\ a_1\ \texttt{dest}\ \{b_1\}] \rangle.\, 0 \\
| \\
\quad (A, B; y\colon C). \\
\quad \texttt{decrypt}\ y\ \texttt{as}\ \{; ym\}_K\ [\texttt{at}\ b_1\ \texttt{orig}\ \{a_1\}]\ \texttt{in}\ 0 \\
)
\end{array}
$$

We use the `at` keyword to attach a label, the cryptopoint, to a specific encryption of decryption operation. Then, the `dest` keyword is used to list those locations where the ciphertext that is created at the associated encryption may be decrypted. Similarly, the `orig` keyword is used to specify by which encryption operation(s) the value we decrypt may have been created. We also allow specifying only a cryptopoint without an `orig` or `dest` clause; this means that any origin or destination is allowed. If no run of the protocol violates the property expressed by any annotation, then we say that the protocol satisfies the specified instance of destination/origin authentication. If, in addition, the annotations were carefully chosen, we may conclude that the protocol is secure.

It seems double work to have to specify both a destination annotation upon encryption and an origin annotation upon decryption. However, when the protocol specification is ran in parallel with a malicious attacker, a ciphertext may be encrypted by the attacker but accepted by a legitimate agent. This breach would not violate the authentication property if only a destination annotation was present. A similar situation holds for the dual case, where the attacker decrypts a ciphertext that was encrypted at a cryptopoint without a destination annotation. Thus, it is usually necessary that we specify both annotations.

Note that in this work, we typically omit annotations completely if they are not relevant to the matter at hand. For example, the semantics of Typed LySa in section 3.2 should be read such that rules containing encryption or decryption may have annotations associated. In section 3.3.4 we will define semantics that do take annotations into account.

3.3.2 SECURITY PROPERTIES

It should now be clear to the reader that destination/origin authentication can be used to specify many common security properties, such as key confidentiality and many forms of authentication. For example, assume that a newly established key is used for transferring an encrypted message, and annotations are added such that only the intended recipient may decrypt the message and that the message may only come from the specified originator. Then, the protocol satisfies key confidentiality if and only if the described annotations cannot be violated. This is the case because if an attacker can obtain the key, he can use it to decrypt the sent message (thus violating the first annotation) or to encrypt new messages with (violating the second annotation).

Sadly, not all security properties can be modelled with destination/origin authentication. The most prominent example is *key freshness*, the guarantee in a key establishment protocol that after a run has completed, the newly established key has indeed been newly generated in that protocol run and has not been in use before. The security risk involved in using a key that is not fresh is that, given enough time, an attacker may be able to discover an old key, as discussed in section 2.1.2. The suggested workaround is to intentionally leak some "old key" to the attacker, as well as all messages that would have been sent with it in a run of the protocol. If the attacker can use this old key to break key confidentiality, the protocol does not guarantee key freshness.

3.3.3 AN EXAMPLE

Our running example can be decorated with destination/origin authentication annotations as follows:

$$
\begin{array}{l}
(\nu\ A)(\nu\ B)(\nu\ S)(\nu\ K_{AS})(\nu\ K_{BS}) \\
(\\
\quad (\nu\ K) \\
\quad \langle A, S, A, \{B, K\}_{K_{AS}}\ [\texttt{at}\ a_1\ \texttt{dest}\ \{s_1\}]\rangle. \\
\quad (\nu\ m) \\
\quad \langle A, B, \{m\}_K\ [\texttt{at}\ a_3\ \texttt{dest}\ \{b_3\}]\rangle. \\
\quad 0 \\
\mid \\
\quad (S, B; y1\colon C). \\
\quad \texttt{decrypt}\ y1\ \texttt{as}\ \{A; yK\colon N\}_{K_{BS}}\ [\texttt{at}\ b_2\ \texttt{orig}\ \{s_2\}]\ \texttt{in} \\
\quad (A, B; y2\colon C). \\
\quad \texttt{decrypt}\ y2\ \texttt{as}\ \{; ym\colon N\}_{yK}\ [\texttt{at}\ b_3\ \texttt{orig}\ \{a_3\}]\ \texttt{in} \\
\quad 0 \\
\mid \\
\quad (A, S, A; z1\colon C). \\
\quad \texttt{decrypt}\ z1\ \texttt{as}\ \{B; zK\colon N\}_{K_{AS}}\ [\texttt{at}\ s_1\ \texttt{orig}\ \{a_1\}]\ \texttt{in} \\
\quad \langle S, B, \{A, zK\}_{K_{BS}}\ [\texttt{at}\ s_2\ \texttt{dest}\ \{b_2\}]\rangle. \\
\quad 0 \\
)
\end{array}
$$

If we also want to test for key freshness, we can leak an "old key" by creating an output operation for this old key and for each of the three messages of the protocol and inserting those after the first line of the above process:

$(\nu\ K_{old})(\nu\ m_{old})$
$\langle K_{old} \rangle.$
$\langle A, S, A, \{B, K_{old}\}_{K_{AS}}\ [\texttt{at}\ i_1] \rangle.$
$\langle S, B, \{A, K_{old}\}_{K_{BS}}\ [\texttt{at}\ i_2] \rangle.$
$\langle A, B, \{m_{old}\}_{K_{old}}\ [\texttt{at}\ i_3] \rangle.$

By communicating these messages onto the ether, we are sure that a Dolev-Yao attacker will be able to read them and by sending the old key in plain text we model explicitly that the attacker has been able to obtain it.

THE ATTACK

We can now easily see that an attack on this protocol is possible. After the messages leaking the old key have been sent, the attacker knows the key K_{old} and the messages sent to establish it. If the attacker then replays the second message, $\langle S, B, \{A, K_{old}\}: K_{BS} \rangle$, agent B will accept this message and correctly decrypt it. B now believes that K_{old} is a valid key, so the attacker can fool B by sending some message $\langle A, B, \{m_{bad}\}: K_{old} \rangle$. B will believe that the message comes from agent A instead of from the attacker and decrypt and use m_{bad}.

Clearly, already when agent B accepts the message replayed by the attacker and decrypts the contained ciphertext, the annotation is violated; the ciphertext was created at cryptopoint i_2 which was not in the set of allowed origin cryptopoints. This means that any automated analysis that can detect the annotation violation at this point will find this attack.

3.3.4 REFERENCE MONITOR SEMANTICS

We can slightly modify the semantics of Typed LySa such that annotations are taken into account. Such semantics obviously do not correspond to any realistic situation, as one would not expect annotated ciphertexts to be physically sent over networks. Nevertheless, by defining semantics such that a process blocks when the annotations are violated, we can be precise about the meaning of these annotations.

Only upon decryption may annotations be violated, as such we only need to redefine the [Decr] rule from section 3.2.2, as shown in Table 3.6:

[DecrRM]	$\dfrac{\ell \in \mathcal{L}' \ \wedge \ell' \in \mathcal{L}}{\begin{array}{c}\texttt{decrypt}\ \{D_0, \ldots, D_{n-1}\}_{D_n}\ [\texttt{at}\ \ell\ \texttt{dest}\ \mathcal{L}] \\ \texttt{as}\ \{D_0, \ldots, D_{k-1}; x_k: T_k, \ldots, x_{n-1}: T_{n-1}\}_{D_n}\ [\texttt{at}\ \ell'\ \texttt{orig}\ \mathcal{L}']\texttt{in}\ P \\ \rightarrow_{RM}\ P\left[x_k \overset{\alpha}{\rightarrow} D_k, \ldots, x_{n-1} \overset{\alpha}{\rightarrow} D_{n-1}\right]\end{array}}$ $\left(\forall_{k \leq i < n-1} T_i = \text{typeof}(D_i)\right)$

Table 3.6. Reference monitor semantics for Typed LySa

Note that the only difference between [DecrRM] and [Decr] is the requirement that the annotations match. If no annotation is specified at all, ℓ resp. ℓ' are taken to be the *unspecified cryptopoint* which cannot be specified in a `dest` or `orig` clause. In case only a `dest` or `orig` clause omitted, we assume $\mathcal{L}$ resp. $\mathcal{L}'$ to be the set of all cryptopoints (which includes the unspecified cryptopoint).

3.4 THE META-LEVEL

We've now seen how to precisely express the behaviour of security protocols as well as the goals we expect them to meet. However, up until now we have only been able to model one run of the protocol, with exactly one agent performing each role. In realistic cases, there are often many agents running the same protocol, often at the same time, in different roles, but communicating with the same server. It is obvious that many attacks can only occur in such a setting that cannot occur in one isolated run. For example, in a flawed protocol, an attacker may use the information received from a run performed by two agents A and B, and use that to fool some other agent, say, C.

Typed LySa includes a meta-language that allows us to precisely specify scenarios such as these. This meta-language is one of the central strengths of (Typed) LySa, in that it allows one to explicitly differentiate between the way different roles, agents and keys are related to one another.

3.4.1 INDEXED OPERATIONS

In short, the meta-language allows us to put subscripts on the | and ν operators, introducing fresh *meta-variables*. For example:

$$|_{i \in X} P$$

means that process P is put in parallel with itself exactly as many times as the amount of elements in X, and in every parallel run of P, i has one of those elements assigned. Similarly,

$$(\nu_{i \in X} a_i)\ P$$

creates as many new names a as the amount of elements in X, each of them indexed by one of those elements. Meta-variables such as i can be used further in a protocol specification as well. For instance,

$$|_{i \in X} (\nu\ K_i) P$$

puts P in parallel with itself $|X|$ times, but in each instance i of P, a unique name K_i is created that is only known to that instance of P. Compare this to

$$(\nu_{i \in X} K_i)\ |_{i \in X} P$$

where each instance of P is able to use every K_i that was created. Finally, we also allow subscripting cryptopoints and variables with available meta-variables. This way, variables and cryptopoints in different parallel runs of the same process expression may be easily distinguished from one another, for example when reading analysis results.

3.4.2 META-LEVEL SETS

Typed LySa meta-variables all range over certain sets, such as X in the example above. Both these variables and their corresponding sets may only exist at the meta-level. However, the semantics of all other operations are only defined in the object-level. To bridge these two worlds, we define an instantiation relation which can be used to transform a meta-level process into an object-level process.

For example, assume the following Typed LySa process:

$$|_{i \in \mathbb{N}} (\nu\ A_i).\langle A_i \rangle.0$$

Here the meta-level elements are the indexed | operator, the meta-variable i and the set $\mathbb{N}$. By instantiating this meta-level process, we get a Typed LySa process with only object-level operations:

$$(\nu\ A_0)\ \langle A_0\rangle.\ 0 \mid (\nu\ A_1).\ \langle A_1\rangle.\ 0 \mid (\nu\ A_2).\ \langle A_2\rangle.\ 0 \mid (\nu\ A_3).\ \langle A_3\rangle.\ 0 \mid \ldots$$

Obviously, the resulting process is infinitely large, which makes automated analysis a little tricky. Nevertheless, because we do not want to lose any generality, we often wish to be able to model such infinitely large situations. Only when performing the analysis, a choice needs to be made about how to transform these infinite processes into finite ones. We combine the best of two worlds by introducing a `let` operation, which we use to give specific sets a name. We can then, in the analysis, consider only a finite subset of such a set. For example:

$$\texttt{let}\ X = \mathbb{N}\ \texttt{in}\ |_{i\in X}(\nu\ A_i).\ \langle A_i\rangle.\ 0$$

Semantically, this corresponds to exactly the same process as the one specified earlier. In its automated analysis, however, we can decide that X correspond to some finite subset X' of $\mathbb{N}$. For example, if X' is {1,2,3}, then we only have the following process to analyse:

$$(\nu\ A_1).\ \langle A_1\rangle.\ 0 \mid (\nu\ A_2).\ \langle A_2\rangle.\ 0 \mid (\nu\ A_3).\ \langle A_3\rangle.\ 0$$

By deciding on which subset X' to use when X is instantiated by a `let` operation, rather than doing so every time X is used, we ensure that the same subset is used every time.

3.4.3 Scenarios

Typed LySa's meta-level can be used to express the relationship between various agents, roles and values. For instance, we can specify whether an agent playing both a role A and a role B in a certain protocol uses the same address for both roles, or not; we can specify whether an agent who shares a master key with a server uses the same master key when playing both roles A and B or a different one for each role. For instance, assume that this master key is the same for any role. In such a case, we might specify a protocol of the following structure:

$$\begin{array}{l}(\nu_{k\in X}\ KM_k)\\(\\\quad |_{i\in X, j\in X} PA_{i,j}\\|\\\quad |_{j\in X, i\in X} PB_{j,i}\\|\\\quad S\\)\end{array}$$

Here, we use $PA_{i,j}$ and $PB_{j,i}$ as a shorthand for "some protocol behaviour for roles A and B", where by convention i identifies the agent playing role A and j identifies the agent playing role B. S is some server, and KM_k is the master key that each agent shares with the server. This scenario therefore specifies that any agent can play both role A and B, can communicate with itself while playing those both roles, and uses the same master key (KM_i or KM_j, which are the same value if $i = j$) no matter which role it is playing.

If we would call our master keys KA_i and KB_j, and use these names respectively in $PA_{i,j}$ and $PB_{i,j}$, then we specify that an agent has a different master key when it plays a different role, for obviously $KA_i \neq KB_j$ even if $i = j$. Similarly, we can specify whether agents have the same

address for different roles or not by using names such as A_i and B_j for identifying agents playing roles A and B, respectively, or by using I_i and I_j in either role.

3.4.4 META-LEVEL SEMANTICS

We define the formal semantics of Typed LySa's meta-level in Table 3.7. We do this by defining an *instantiation relation*, which describes how a Typed LySa process that includes meta-level elements can be converted to a (generally much longer) Typed LySa process that only contains object-level constructs. The object-level semantics can then be applied to the process, precisely defining its behaviour. The instantiation relation is of the following form:

$$\Gamma, \mathcal{N} \vdash L \Rrightarrow P$$

Here, L is a meta-level process and P is a (possibly infinite) object-level process. We use two context components, Γ and $\mathcal{N}$, where Γ is a partial mapping of set identifiers to the actual sets they represent, and we write $\Gamma[X \mapsto S]$ and $\Gamma(S)$ for updating and querying Γ, respectively. $\mathcal{N}$ is a set that collects the free names of the process P.

[Let]	$\dfrac{\Gamma[X \mapsto S'], \mathcal{N} \vdash L \Rrightarrow P}{\Gamma, \mathcal{N} \vdash \texttt{let}\ X = \mathbb{S}\ \texttt{in}\ L \Rrightarrow P}\ (S' \subseteq_{\text{fin}} \mathbb{S})$
[INew]	$\dfrac{\Gamma, \mathcal{N} \cup \{\alpha_{i_0,\dots,i_{k-1}\overline{a_j}} \mid 0 \le j < m\} \vdash L \Rrightarrow P}{\Gamma, \mathcal{N} \vdash \left(\nu_{i_k \in X_k,\dots,i_{n-1} \in X_{n-1}}\ \alpha_{i_0,\dots,i_{n-1}}\right) L \Rrightarrow \left(\nu\ \alpha_{i_0,\dots,i_{k-1}\overline{a_0}}\right) \cdots \left(\nu\ \alpha_{i_0,\dots,i_{k-1}\overline{a_{m-1}}}\right) P}\ \left(\begin{matrix}\{\overline{a_j} \mid 0 \le j < m\} = \Gamma(X_k) \times \dots \times \Gamma(X_{n-1}) \\ (0 \le k \le n)\end{matrix}\right)$
[IPar]	$\dfrac{\begin{matrix}\Gamma, \mathcal{N} \vdash L[i_h \mapsto a_{0,h} \mid 0 \le h < k] \Rrightarrow P_0 \\ \vdots \\ \Gamma, \mathcal{N} \vdash L\langle i_h \mapsto a_{m-1,h} \mid 0 \le h < k\rangle \Rrightarrow P_{m-1}\end{matrix}}{\Gamma, \mathcal{N} \vdash \mid_{i_0 \in X_0,\dots,i_{k-1} \in X_{k-1}} L \Rrightarrow P}\ \left(\begin{matrix}\{\overline{a_j} \mid 0 \le j < m\} = \Gamma(X_0) \times \dots \times \Gamma(X_{k-1}) \\ 0 < k\end{matrix}\right)$
[Onew]	$\dfrac{\Gamma, \mathcal{N} \cup \{\alpha\} \vdash L \Rrightarrow P}{\Gamma, \mathcal{N} \vdash (\nu\ \alpha)\ L \Rrightarrow (\nu\ \alpha)\ P}$
[OPar]	$\dfrac{\Gamma, \mathcal{N} \vdash L_1 \Rrightarrow P_1 \quad \Gamma, \mathcal{N} \vdash L_2 \Rrightarrow P_2}{\Gamma, \mathcal{N} \vdash L_1 \mid L_2 \Rrightarrow P_1 \mid P_2}$
[Proc]	$\dfrac{}{\Gamma, \mathcal{N} \vdash P \Rrightarrow P}$

Table 3.7. Meta-level to object-level instantiation relation, $\Gamma, \mathcal{N} \vdash L \Rrightarrow P$

In the first rule in Table 3.7, (Let), we choose a finite subset S' of the specified set $\mathbb{S}$. We map the set name X to this S' in the Γ component of the context to ensure that the same subset is used for all occurrences of X.

In [INew], we formalise the meaning of the indexed restriction operator. Because this operator can declare its own meta-variables to be used in place but also use meta-variables that have been defined in an earlier indexed parallel operator, the rule is quite complex. The meta-variables $i_0 \dots i_{k-1}$ have been defined earlier, and $i_k \dots i_{n-1}$ are the ones declared in place. This means that if $k = n$, only one new name is made, but if $k < n$, a name must be declared for each value that $i_k, \dots, i_{n-1}$ can take, so we must take the Cartesian product of the corresponding sets $X_k \dots X_{n-1}$. We denote the resulting list of m vectors of meta-values by $\overline{a_0} \dots \overline{a_{m-1}}$.

Rule [IPar] defines indexed parallelism, which makes new meta-variables $i_0 \dots i_{k-1}$ available in its operand process L. However, we immediately instantiate a copy of L for each

combination of values that the $i_0 \dots i_{k-1}$ may be made equal to. We do this by taking the Cartesian product of the corresponding sets $X_0 \dots X_{k-1}$ and for each vector of meta-values $\overline{a_j}$ found, we create an instance of L with all occurrences of $i_0 \dots i_{k-1}$ substituted by the corresponding values of in $\overline{a_j}$, denoted by $L[i_h \mapsto a_{j,h}]$, $0 < h \leq k$.

The rule [Onew] simply transfers a non-indexed restriction operator to the object-level. Similarly, [OPar] transfers a non-indexed parallel operator to the object-level, keeping the context variables in tact. [Proc] retains any process consisting only of object-level elements.

3.5 THE ATTACKER ENTRY-POINT

In addition to exactly specifying the scenario in which our protocol may run, we wish to also precisely control which information is available to which parties. Most specifically, it is important that we make clear which names can be known by outside parties not directly involved in the protocol, possibly being malicious attackers.

Recall that in Typed LySa, we require that each name is declared with a ν operator before it is used. We introduce the • symbol, which specifies the point in the protocol where it is *open to attack*. We need to insert such a point in order to be able to differentiate between the names that we only expect to be known by honest agents, and names which are expected to be general knowledge, meaning that the attacker may use them to forge an attack. In our analysis, we insert the attacker process exactly at the • symbol, meaning that the names that are known at that location in the protocol description define the initial knowledge of the attacker. We elaborate on this in Chapter 5.

Typically, we would model our protocol so that the addresses of the communicating agents are universally known, whereas master keys that agents share with the server are modelled to be known only by legitimate protocol participants. The general structure of a Typed LySa protocol description, therefore, often follows the following outline:

$$
\begin{array}{l}
(\nu\, u_1) \dots (\nu\, u_n) \\
(\\
\quad \bullet \\
\mid \\
\quad (\nu\, p_1) \dots (\nu\, p_m) \\
\quad (P_A \mid P_B \mid \dots \mid P_S) \\
)
\end{array}
$$

Here, $u_1 \dots u_n$ are universally known names and $p_1 \dots p_m$ are names only known by the processes $P_A \dots P_S$, which model the behaviour of the honest agents interacting in the protocol.

3.5.1 SEMANTICS

The attacker entry-point requires the addition one more rule to the meta-level semantics of section 3.4.4, which is specified in Table 3.8.

[Open]	$\overline{\Gamma, \mathcal{N} \vdash \bullet \Rrightarrow P'}$	for an arbitrary P' such that there are no free variables in P' and all free names in P come from $\mathcal{N}$.

Table 3.8. The meta-level to object-level instantiation relation for the attacker entry-point

With the rule [Open] we define that at the dot we may insert any process that only makes use of the variables and names available in its scope (and any new ones it may declare itself). A Dolev-Yao attacker typically is such a process.

3.6 SUMMING UP

3.6.1 GRAMMAR OF TYPED LYSA

We propose a complete grammar for Typed LySa. This grammar describes the complete syntax of Typed LySa, that is, it includes both the object-level and the meta-level as well as annotations.

I ::= $\{i_0, \dots, i_{n-1}\} \mid \epsilon$
$IDef$ ::= $\{i_0 \in S_0, \dots, i_{n-1} \in S_{n-1}\}$
CP ::= c_I
E ::= $a_I \mid x \mid \{E_0, \dots, E_{n-1}\}_{E_n}$ [at CP_0 dest $\{CP_1, \dots, CP_m\}$]
T ::= $N \mid C$
P ::= let $S = \mathbb{S}$ in P
$\mid \langle E_0, \dots, E_{n-1} \rangle . P$
$\mid (E_0, \dots, E_{k-1}; x_k : T_k, \dots x_{n-1} : T_{n-1}) . P$
$\mid$ decrypt E as $\{E_0, \dots, E_{k-1};\ x_k : T_k, \dots x_{n-1} : T_{n-1}\}_{E_n}$ in P [at CP_0 orig $\{CP_1, \dots, CP_m\}$]
$\mid (\nu\ a)\ P$
$\mid (\nu_{IDef}\ a_I)\ P$
$\mid |_{IDef} P$
$\mid P_1 \mid P_2$
$\mid \bullet$
$\mid 0$
$(0 \leq k \leq n \wedge 0 < n)$ *must hold in each rule*

Table 3.9. The grammar of Typed LySa

In Table 3.9, an x represents a variable, an a represents a name, a c represents a crypto-point, an i represents a meta-variable and an S represents a set. $\mathbb{S}$ can be any finite or infinite set, expressed in standard mathematical notation. All other identifiers are constants or defined in the grammar itself.

3.6.2 WELL-FORMEDNESS RESTRICTIONS

The syntax of Typed LySa is rather liberal, and allows an author to express a number of constructs that make analysis overly complex. In order to address this, we define a number of well-formedness restrictions:

1. Newly created variables must be indexed by all currently available meta-variables.
2. Newly bound names must be indexed by all currently available meta-variables.
3. Any name may be bound only once in the whole process.
4. A top-level process may not contain any free variables or free names.
5. Only literal ciphertexts and variables of type C may occur on the left hand side of the decrypt operation.

The first three restrictions ensure that an identifier used for any name or variable is always unique in the process, in other words, no two parallel runs of a subprocess can bind a

variable or name with the exact same identifier. These restrictions have a double purpose: first of all, we believe that they make a process easier to read and prevent mistakes (as many typing mistakes may be detected as well-formedness breaches) and secondly, they make analysis significantly easier.

ANALYSIS RESTRICTIONS

In order for our analysis to be possible, we have to add a few more restrictions to the form of the process.

6. There may be no more than one • in the process specification
7. This • must occur before any input, output decryption or indexed parallelism is performed.

These two restrictions are necessary to make conversion to LySa (for analysis with the LySa tool) possible. Additionally, it will make conversion to mCRL2 more straightforward, and the conversion rules in Chapter 5 depend on it.

3.6.3 THE EXAMPLE COMPLETED

We now present a complete specification of our running example, a variation of the Wide Mouthed Frog protocol. In this scenario, we assume agents playing a different role have a different address (A_i or B_j) and that they share a different master key (KA_i and KB_j) with the server. We consistently use the meta-variable i to identify names belonging to role A and j to identify names that belong to role B. Any name, variable or cryptopoint that is indexed by both i and j is unique to the protocol run that attempts to establish a new key for use between agents A_i and B_j.

$$
\begin{array}{l}
\texttt{let } X = \mathbb{N} \texttt{ in} \\
(\nu_{i\in X}\ A_i)(\nu_{j\in X}\ B_j)(\nu\ S) \\
\quad \bullet \\
\mid \\
(\\
\quad (\nu_{i\in X}\ KA_i)(\nu_{j\in X}\ KB_j) \\
\quad\quad (\mid_{i\in X, j\in X} \\
\quad\quad\quad (\nu\ K_{ij}) \\
\quad\quad\quad \langle A_i, S, A_i, \{B_j, K_{ij}\}_{KA_i}\ [\texttt{at } a1_{ij}\ \texttt{dest}\ \{s1_{ij}\}]\rangle. \\
\quad\quad\quad (\nu\ m_{ij}) \\
\quad\quad\quad \langle A_i, B_j, \{m_{ij}\}_K\ [\texttt{at } a3_{ij}\ \texttt{dest}\ \{b3_{ij}\}]\rangle. \\
\quad\quad\quad 0\ ) \\
\quad \mid \\
\quad\quad (\mid_{j\in X, i\in X} \\
\quad\quad\quad (S, B_j; y1_{ij} : C). \\
\quad\quad\quad \texttt{decrypt } y1_{ij}\ \texttt{as}\ \{A_i; yK_{ij} : N\}_{KB_j}\ [\texttt{at } b2_{ij}\ \texttt{orig}\ \{s2_{ij}\}]\ \texttt{in} \\
\quad\quad\quad (A_i, B_j; y2_{ij} : C). \\
\quad\quad\quad \texttt{decrypt } y2_{ij}\ \texttt{as}\ \{; ym_{ij} : N\}_{yK_{ij}}\ [\texttt{at } b3_{ij}\ \texttt{orig}\ \{a3_{ij}\}]\ \texttt{in} \\
\quad\quad\quad 0\ ) \\
\quad \mid \\
\quad\quad (\mid_{i\in X, j\in X} \\
\quad\quad\quad (A_i, S, A_i; z1_{ij} : C). \\
\quad\quad\quad \texttt{decrypt } z1_{ij}\ \texttt{as}\ \{B_j; zK_{ij} : N\}_{KA_i}\ [\texttt{at } s1_{ij}\ \texttt{orig}\ \{a1_{ij}\}]\ \texttt{in} \\
\quad\quad\quad \langle S, B_j, \{A_i, zK_{ij}\}_{KB_j}\ [\texttt{at } s2_{ij}\ \texttt{dest}\ \{b2_{ij}\}]\rangle\ .
\end{array}
$$

0)
)

If we had wanted to specify that agents have only one address, independent of their role, we would replace each occurrence of both A_i and B_j by I_k and declare I_k as $(\nu_{k \in X}\ I_k)$. Similarly, if we want to specify that agents share the same master key KM with the server independent of their role, we would replace each KA_i and KB_j by KM_k and declare KM accordingly. In our analysis, we can scale the protocol up or down by adjusting the size of the subset of $\mathbb{N}$ that we equate X to.

3.7 LySA

Typed LySa is only a minor variation of LySa, a process calculus designed by Bodei *et al.* in 2004 [**2**] for the static analysis of security protocols.

3.7.1 Removing type specifiers

The most prominent difference between LySa and Typed LySa is, unsurprisingly, that in Typed LySa we require that types of variables are specified in input and decryption operations. We do so because it makes conversion to mCRL2 much simpler, and additionally provides more opportunities for optimising the state space generation process.

This difference is smaller than it seems: only in marginal cases will a protocol be secure when types are specified but flawed when they are not. In typical protocols, ciphertexts are decrypted soon after they are received and also in LySa, decryption fails if the value that is decrypted is not a ciphertext. Nevertheless, the additional typing assumption has real consequences and exposes flaws in strictly fewer protocols. For example, consider the following three parallel processes expressed in Typed LySa:

$$\langle \{Q\}_K \rangle.\, 0 \mid (; x{:}N).\, \langle R, x \rangle.\, 0 \mid (R; y{:}C).\, \texttt{decrypt}\ y\ \texttt{as}\ \{; y' {:} N\}_K\ [\texttt{at}\ c\ \texttt{orig}\ \{c'\}]\ \texttt{in}\ 0$$

The leftmost process will send some name Q encrypted with some (universally known) key K. No process is ready to receive this message, however, because only the middle process is expecting just one value, but that must be a name and not a ciphertext. As such, the process blocks. Now, consider the same process in LySa:

$$\langle \{Q\}_K \rangle.\, 0 \mid (; x).\, \langle R, x \rangle.\, 0 \mid (R; y).\, \texttt{decrypt}\ y\ \texttt{as}\ \{; y'\}_K\ [\texttt{at}\ c\ \texttt{orig}\ \{c'\}]\ \texttt{in}\ 0$$

In this situation, the middle process will happily accept the ciphertext and forward it to the rightmost process. This process will attempt to decrypt it, but the cryptopoint c' was not specified upon encryption, thus a flaw is exposed.

Clearly, a protocol like this is not very realistic: the fact that the middle process accepts and forwards a name suggests that some fourth process is probably the intended recipient of that message. Every set-up in which the additional typing assumption leads to additional flaws that we have been able to find is of this nature: one agent forwards a name untouched, which is then intercepted by an agent for whom the message was not intended and who thinks that it is a ciphertext.

Such a flaw is a type flaw not unlike many other type flaw attacks, most of which also in LySa cannot be captured. Furthermore, we underline that both LySa and Typed LySa assume that every value inside a message can be securely distinguished from one another. Most

implementation methods that safely ensure such a situation (can) provide enough redundancy and information that the type of each value can also be determined. This means that most practical situations that can be appropriately modelled in LySa can be described in Typed LySa equally well.

3.7.2 Differences on the meta-level

LySa does not support the concept of the attacker entry-point, •. Instead, this symbol was ported into Typed LySa from LySa^{NS} [**5**], a slightly stronger but more complex version of LySa. In LySa, the entire given process runs in parallel with an attacker, and all free names in this process are assumed to be universal knowledge, so exactly these names are made to be the attacker's initial knowledge.

For Typed LySa, we choose instead to explicitly specify the information known to the attacker. This is done for two reasons. First of all, it makes the conversion to other languages more straightforward, as no explicit detection of free names has to be performed. Secondly, it allows us to demand that every name that is used must be declared somewhere – globally free names are not permitted. By requiring this, mistyped names do not suddenly become part of the attacker's knowledge, often resulting in mysterious flaw reports. Instead they result in a parse error.

Finally, there is one cosmetic difference that we allow the indexed parallel operator to have multiple declared indices in one go. In other words, $|_{i\in S, j\in T} P$ in Typed LySa is written $|_{i\in S}|_{j\in T} P$ in LySa. We allow this for brevity and because it results in slightly more concise and simple mCRL2 specifications when converted.

3.7.3 The LySa Tool

The LySa calculus has been designed alongside a static analysis that can find flaws in LySa processes. This analysis has been implemented in the LySa tool. In this section, we briefly describe how this analysis works. Because we do not use any of the output of the LySa tool in this work, the LySa analysis will only be discussed informally. An in-depth discussion of the LySa analysis is presented in [**4**] and [**2**].

LySa's static analysis keeps track of the set of ciphertexts that are successfully decrypted at each relevant point. The behaviour of the protocol is over-approximated by mapping each name and variable to a *canonical* name or variable. This means that multiple names may in fact map to the same canonical name. For instance, each name which corresponds to the same identifier (where meta-indices are, however, taken into account) is assigned the same canonical name. The canonical name of some name n in written $\lfloor n \rfloor$. In a process such as $(\nu_{i\in\{1,2\}} A_i)((\nu\, B)\, P \mid (\nu\, B)\, P')$, $\lfloor A_1 \rfloor \neq \lfloor A_2 \rfloor$ but both B's map to the same $\lfloor B \rfloor$ even though they are semantically different names.

Using these canonical names and variables, the analysis keeps track of three sets, ρ, κ and ψ. Set ρ maps canonical variables to a set of all the canonical values they may be bound to. κ keeps track of all the messages (tuples of canonical values) that are ever sent on the network. Finally, ψ contains all possible annotation flaws. The protocol source is then explored and for every message sent, κ grows. For every input operation encountered that matches one or more tuples in κ, ρ is expanded such that the variables declared in the input operation may be bound to these names. A similar thing happens upon decryption, where all the possible values of the decrypted variable (collected in ρ) are matched against `decrypt`'s second operand and ρ is expanded upon success. If this is the case, the annotation attached to the decrypted

ciphertext is matched against the decryption annotation and if this fails, ψ is expanded with a tuple of the encryption and decryption crypto-point . If the analysis has finished, and ψ is not empty, then there may be flaws.

Notice how this analysis clearly over-approximates the behaviour of a protocol: for example, any element in κ is matched against a fitting input operation, even if a protocol has been designed such that certain of these matches can never occur. Nevertheless, only one protocol) is known where the analysis reports false positives (the Yahalom protocol and variations of it, [**7**])

Instead of including an attacker process that acts precisely like the Dolev-Yao attacker, a much simpler attacker is included that produces exactly the same *analysis results* as a real Dolev-Yao attacker. This works because any process that will have the same impact on the ρ, κ and ψ sets as a Dolev-Yao attacker would, will produce the same outcome.

The output of the LySa tool are exactly these three sets. Because ρ and κ can grow tremendously if only sets of canonical names (and ciphertexts built from these) would be collected, names are represented using tree grammars instead, which allows the tool to keep track of these values with maximal sharing. As a result, the tool is reasonably fast: analysis completes on most protocols in at most a few seconds.

Chapter 4 mCRL2

In this chapter we introduce mCRL2, a specification language for communicating processes, and its associated toolset. mCRL2 is a powerful language, capable of describing a large variety of processes, protocols and behaviour. For conciseness, we will only describe those parts of the mCRL2 language that we will use in finding attacks on security protocols. Most importantly, we will disregard mCRL2's support for timed processes. Because Typed LySa does not support any notion of time, considering it in mCRL2 makes the discussion of mCRL2 processes needlessly complex.

It should be noted that the semantics of every mCRL2 process is formally defined by means of a structural operational semantics. Additionally, mCRL2 comes with a large set of axioms defining equalities between syntactically different process expressions; two processes are considered equal if they are bisimulation equivalent. As these axioms may be used to manipulate processes or prove properties about them, it should be clear that also without the toolset, mCRL2 is a powerful formalism for specifying and studying the behaviour of processes.

In this work, the formal semantics that define mCRL2 nor its axioms will be discussed. Instead, we limit ourselves to a more informal discussion of those elements of the mCRL2 language that will be used in later chapters. For an in-depth discussion of mCRL2 as well as its formal definition, please see [**14**].

4.1 Process expressions

The primary building block of an mCRL2 process is the *action*. An action represents an atomic, observable event, expressing that the system that we model does something specific. Because mCRL2 can be used to model a wide variety of systems, including robot controllers, communication protocols, automated parking lots, online shops and puzzle games, the nature and names of actions can also wildly vary. Typical action names include `send`, `receive`, `shutdown_robot`, `a`, `b`, `place_order`, etcetera.

Actions are atomic. This means that if at some point both action a and action b can happen, a completely happens before b or vice versa. There is one exception, which is the situation where a and b happen at exactly the same time. This is written $a|b$ and we call such an action a *multi-action*.

There are two special actions, δ and τ. δ is the *deadlock*, which represents a situation in which no action can occur. τ is the *empty multi-action*, which is unobservable.

4.1.1 From actions to processes

Actions can be combined to form processes. A process describes the behaviour of one or more principals that act according to certain rules. Using the period, one can specify that one action is followed by another action, called sequential composition: $a \cdot b$ means that first a occurs, and then b. The plus sign is the choice operator: it specifies the nondeterministic choice between either of its operands, also called alternative composition. So, $a + b$ means that either a or b can occur, but not both. Using parentheses, we can combine these two operators to define simple processes: $(a \cdot b) + (c \cdot d)$ expresses that either a followed by b can occur, or c followed by d. Compare this to $a \cdot (b + c) \cdot d$, which specifies that a always occurs, is followed by either b or c, which is then inevitably followed by d. Sequential composition has a higher priority than alternative composition, so the former process could also be written as $a \cdot b + c \cdot d$.

Parameters and sums

mCRL2 actions can be associated with a *parameter*: a tuple of data expressions. These data expressions are described in more detail in section 4.2, but for now it suffices to know that all common data types such as Booleans, integers, natural numbers, real numbers, etcetera are supported, along with their standard operators. A parameter is bound to an action with parentheses, e.g. $a(1,2,3)$ or `set_error_flag`$(true)$.

The presence of data allows us to generalise the choice operator, $+$, over the domain of a certain type, written as a sum expression. For example, $\sum_{c:\mathbb{B}} a(c)$ is equal to $a(true) + a(false)$. Similarly, $\sum_{n:\mathbb{N}} b(n)$ could be expanded to $b(0) + b(1) + b(2) + \ldots$, but this is not a finite process expression. Notice that the sum operator creates one or more new variables which are bound in the associated process. This is one of two ways to define variables in mCRL2 process expressions.

Deterministic choice based on data can be expressed with the conditional operator. For a Boolean expression c and processes p and q, writing $c \to p \diamond q$ means that if c is $true$, p will occur, and otherwise q will occur. As a shorthand, the last part of the conditional operator may be omitted, which means that a process blocks if the condition is *false*. In other words, $c \to p$ is equal to $c \to p \diamond \delta$.

Parallelism, communication and abstraction

Multiple processes can be modelled to run at the same time using the parallel operator, $\|$. This operator specifies that actions from its two operand processes happen independently from one another. Thus, they can be interleaved in any order, or occur simultaneously and form a multi-action. For example, two single actions put in parallel, $a \| b$, is equal to $a \cdot b + b \cdot a + a|b$. The complexity of a process and the amount of ways in which it can be run increase tremendously when more complex processes are put in parallel with one another. For example, $a \| (b \cdot c)$ is equal to $a \cdot b \cdot c + b \cdot (c \cdot a + a \cdot c + a|c) + (a|b) \cdot c$.

mCRL2 has no direct support for high-level communication, channels, and similar constructs often found in process calculi. Instead, two operators, Γ and ∇, are introduced which allow us

to very precisely model the method of communication that may occur. Γ is the *communication operator*, which takes a multi-action and renames two actions inside it to a different single action, but only if the two actions have exactly corresponding data parameters. For example, $\Gamma_{\{a|b\to c\}}(a(1)|b(1))$ is equal to $c(1)$. Per comparison, $\Gamma_{\{a|b\to c\}}(a(1)|b(2))$ and $\Gamma_{\{a|b\to c\}}(a(1))$ are both equal to $a(1)|b(2)$ and $a(1)$, respectively; in these cases, the communication operator does nothing. The semantics of Γ have been defined so that it can propagate down process expressions: $\Gamma_{\{a|b\to c\}}(d\cdot(a(1)|b(1)))$ is equal to $d\cdot\Gamma_{\{a|b\to c\}}(a(1)|b(1))$, which in turn equals $p\cdot c(1)$.

The second operator, ∇, is the allow operator. This operator blocks any action from occurring except those specified (and τ). For instance, $\nabla_{\{c\}}(a+b+c)$ is equal to c. We can now model synchronous communication by a combination of these two operators. Consider two actions, s and r, both having a natural number as parameter. The process $\nabla_{\{c\}}(\Gamma_{\{s|r\to c\}}(s(1)\|r(1)))$ is now equal to $c(1)$, despite the complexity of the parallel operator; s and r actions are not allowed to occur alone.

When combining this setup with the sum operator, we can model real synchronous communication, where the receiving party does not know which value to expect:

$$\nabla_{\{c\}}(\Gamma_{\{s|r\to c\}}(s(1)\|\textstyle\sum_{n:\mathbb{N}} r(n)))$$

Despite the sum expression over an infinite domain in the above process, the process is finite because all occurrences of r will be blocked except for $r(1)$ which is renamed to $c(1)$, when it synchronises with $s(1)$.

Finally, the τ operator can be used to *hide* certain behaviour, by renaming specified actions to the empty multi-action. This can be used to mark certain actions as internal behaviour, which is unobservable and in which we are not interested. Hiding selected actions can severely reduce the complexity of a process, therefore potentially improving the speed of automated analysis tremendously. As an example, $\tau_{\{b\}}(a\cdot b\cdot c)$ equals $a\cdot\tau\cdot c$.

4.1.2 Named processes and recursion

Process expressions can be given a name with the **proc** keyword. The process mentioned above could be called A by means of the following expression:

proc $A(m{:}\,\mathbb{N}) = \nabla_{\{c\}}\Gamma_{\{s|r\to c\}}s(m)\|\sum_{n:\mathbb{N}} r(n)$;

Such a named process can be invoked from other processes. For example $a\cdot A(1)$ is equal to $a\cdot c(1)$, because the process $A(1)$ is equal to $c(1)$. Named processes are the second place in mCRL2 where variables may be bound, and they are valid inside the whole associated process.

More interestingly, however, a process can be called from itself, allowing for recursion. A machine that endlessly shoots tennis balls in a fixed direction, for instance, could be modelled as

proc $A =$ `shoot_tennis_ball` $\cdot A$;

The above is not a complete mCRL2 specification: we need to declare every used action and the data type of its parameters using the **act** keyword. Additionally, by means of the obligatory **init**

keyword we specify the entry-point of our process. The endless tennis ball shooter can be modelled as a valid mCRL2 specification as follows:

act `shoot_tennis_ball`;
proc $A =$ `shoot_tennis_ball` $\cdot A$;
init A;

Note that it is perfectly valid to specify a whole process expression after **init**, and not use any **proc** keyword at all. That situation disallows recursion, however.

Now consider the following specification:

act $s, r, c: \mathbb{N}$;
proc $S(m: \mathbb{N}) = s(m) \cdot S(m+1)$;
proc $R = \sum_{n:\mathbb{N}} (n < 5) \rightarrow (r(n) \cdot R)$;
init $\nabla_{\{c\}}\Gamma_{\{s|r\rightarrow c\}}(S(0) \| R)$;

Here, we may view processes S and R as two communicating parties, one which sends a series of increasing numbers and another which receives those numbers. Because R blocks when no value lower than 5 can be received, this process is finite and is equal to $c(0) \cdot c(1) \cdot c(2) \cdot c(3) \cdot c(4)$.

This final example should illustrate that it is possible to clearly express all kinds of protocols and communicating systems in mCRL2.

4.2 DATA EXPRESSIONS

In the last example we saw how a condition can be used together with parameterised actions to let data influence process behaviour. In mCRL2 one can define data types and functions of arbitrary complexity. This way, many types of information may be communicated and this information may be manipulated or used to control process execution in many ways.

4.2.1 DATA TYPES

mCRL2 has a number of built-in elementary data types, as well as constructions to make user-defined *sorts* which may have a more complex structure.

The elementary data types are $\mathbb{B}, \mathbb{N}^+, \mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$, representing the Booleans, the positive natural numbers, the natural numbers, the integers and the real numbers, respectively. Most common logical and arithmetic operators are defined, and values of these types are guaranteed to have a unique representation, so that we can check for equality and inequality. Given the variables $n: \mathbb{N}, i: \mathbb{Z}, b: \mathbb{B}$, a few examples of valid expressions over these types are given in Table 4.1.

$b \Rightarrow (n \approx \text{abs}(i))$	($true$ iff b holds then n must be equal to the absolute value of i)
$b \approx (6 + i) < (n/2)$	($true$ iff b holds if and only if $6 + i$ is smaller than $n/2$)
$if(b, i, i + 1)$	(equals i if b holds, or $i + 1$ otherwise)

Table 4.1. Some mCRL2 data expressions

Note that $\approx$ is object-level equality, an operator in the mCRL2 language, whereas $=$ is meta-level equality.

More complex data types available in mCRL2 are sets, bags, functions, lists and structs, of which we will introduce the latter two here. For any sort A, the sort of lists over A is made by writing $List(A)$. Literal lists can be specified using square brackets, for example $[1, 2, 4]$ is an instance of sort $List(\mathbb{N})$. Additional operations on lists include prepending or appending elements, concatenating lists, finding the element at a specified offset or determining the length of the list. For example, $(4 \triangleright [1, -5]) ++ m$ with m: $List(\mathbb{Z})$ prepends 4 to the list $[1, -5]$ and concatenates m to the result.

Structs in mCRL2 structural data types like they exist in many functional programming languages. For instance,

sort `Tree` $=$ **struct** `leaf`$(\mathbb{N})$ | `node`(`Tree`, `Tree`);

specifies a tree over the natural numbers. Here, `leaf` and `node` are constructors that create a specific instance of the tree, for instance

`node`(`node`(`leaf`(1), `leaf`(2)), `leaf`(2))

mCRL2 supplies a shorthand method for specifying accessor functions and recogniser functions, as per the following example:

sort `Tree` $=$ **struct** `leaf`(`value`: $\mathbb{N}$)? `is_leaf` | `node`(`left`: `Tree`, `right`: `Tree`)? `is_node`;

Assuming that a: `Tree` is a variable that contains the tree literal specified above, we can write the following expressions:

`value`(`right`(a))	(equals 3)
if(`is_leaf`(`left`(a)), `value`(`left`(a)), -1)	(equals -1 because the left hand side of a is a node)

Any constructor function can have zero or more data members of arbitrary types. As such, these structured types can also be used for making simple enumerated types or for allowing one to mix multiple types in one sort:

sort `MonkeyWants` $=$ **struct** `banana` | `apple` | `typewriter`;
sort `IntBool` $=$ **struct** `i`(`int`: $\mathbb{Z}$)? `is_int` | `b`(`bool`: $\mathbb{B}$)? `is_bool`;
sort `IntBoolList` $=$ $List$(`IntBool`);

Structs and lists are the only complex mCRL2 types that are used in this project. For some of our purposes, sets would in fact be more appropriate, but the mCRL2 implementation of sets as of July 2008 suffers from the drawback that sets that may be identical cannot always be

distinguished as such. For instance, the empty set to which an element is added and subsequently removed is not equal to the empty set.

4.2.2 MAPPINGS

mCRL2 has a powerful equation system, which allows a user to easily specify complex auxiliary functions over data parameters. An example:

map height: Tree → ℕ;
var s, t: Tree;
n: ℕ;
eqn height(node(s, t)) = max(height(s), height(t)) + 1;
height(leaf(n)) = 1;

Here we define a function `height` that takes a `Tree` and returns a natural number. In a way similar to functional programming languages, the function is recursively defined in the structure of the `Tree` type: either of the two equations can, on the left hand side, be applied to any `Tree` literal, yielding strictly smaller trees on the right hand side.

Besides pattern matching on the structure of data parameters, we can define conditional equations. For example, `height` could also be defined as follows:

map height: Tree → ℕ;
var t: Tree;
eqn (is_node(t)) → height(t) = max(height(left(s)), height(right(t))) + 1;
(is_leaf(t)) → height(t) = 1;

The semantics of both definitions are equal, and which notation to use depends on the situation and on taste.

4.3 THE MCRL2 TOOLSET

The mCRL2 toolset consists of a large variety of tools that work on process specifications. In this section we will briefly discuss those tools that are most relevant to this work.

Most mCRL2 tools are command-line tools operating on files. There is also an interactive environment, SQuADT, which combines the power of many tools into one application, as well as some visualisation and simulation tools that have a graphical user interface. We will only treat selected command-line tools, however.

4.3.1 THE LINEARISER

The lineariser tool, `mcrl22lps`, is (currently) the only tool that directly operates on mCRL2 specifications. Its main purpose is translating an mCRL2 specification to a *linear process specification* (LPS), which specifies an mCRL2 process of a restricted form. This form, the *linear process definition*, is a series condition-action-effect rules. They are specified as a list of alternatively composited summands in each of which exactly one action can occur, guarded by some condition and directly followed by a recursive invocation of the process itself. Both the data parameter of the action and the recursive invocation can have a different operations on

the associated data parameters. Linear process definitions do not contain any parallelism or communication and blocking operators, which makes them an easy format to manipulate.

For example, the following is a linear process definition:

proc $P(c{:}\,\mathbb{B}, n{:}\,\mathbb{N}) = c \to a(n).\,P(\neg c, n) + \neg c \to b(n).\,P(\neg c, n+1)$;

Notice how the variable c is only used to determine which of the two clauses is chosen, and made to alternate at every recursive process invocation. As such, $P(true, 0)$ performs $a(0) \cdot b(0) \cdot a(1) \cdot b(1) \cdot a(2) \cdot \ldots$, so $P(true, k)$ is equal to $Q(k)$ in following (non-linear) process definition:

proc $Q(n{:}\,\mathbb{N}) = a(n).\,b(n).\,P(n+1)$;

Any mCRL2 process specification can be transformed into a linear process specification, which is what `mcrl22lps` does. All other tools, then, operate on LPSs instead of mCRL2 specifications. Because of the restricted form of LPSs, manipulating and analysing them is significantly simpler. As such, the mCRL2 toolset also provides a library interface for directly manipulating LPSs.

4.3.2 Generating labelled transition systems

The most common representation format of process behaviour is without doubt the labelled transition system (LTS). An LTS is a directed graph where each node is a state and each arc is labelled by the (multi-)action that is performed to get to another state. Because an LTS explicitly contains every state in which a process can be, its size is a good measure for the *state space* of a process. Because of this, we often use the terms LTS and state space interchangeably.

The mCRL2 toolset can generate an LTS from an LPS through the `lps2lts` tool. This tool is the main tool that we use in this work. The transition systems generated are often very large, and running lps2lts may take a lot of time. As such, one of our central challenges is modelling a system in such a way that the state space stays within limits.

The LTS generator has a few features that we thankfully use. Most importantly, it can be configured to scan for a certain action to occur. If it this happens, we can save a *trace* to that action to a trace file. This trace, the series of actions taken until the specified action was occurred, can then be pretty-printed for human readability or replayed using a simulation tool. As we will see in Chapter 5, we model one action, FAIL, specifically for use with this feature. This action will occur if and only if a flaw in the protocol is found. The only actions that may otherwise occur represent messages being sent, which means that a trace is in fact an instance of an attack on the protocol.

Because it is, in general, only these traces that we are interested in, one can safely remove the generated LTS file after the tool has completed.

4.3.3 Model checking with PBESs

The way of doing model checking with the mCRL2 toolset involves combining an LPS with a modal formula to create a Parameterised Boolean Equation System (PBES) [**13**]. This PBES can then be solved using a solver tool, and we can determine whether or not the modal formula

holds for this LPS. The PBES technique is an effective method for model checking and is in many cases more efficient than generating the whole state space with the LTS generator, because sometimes whole parts of a process can be left unexplored.

However, in our case this does not suffice, because there currently is no straightforward way for obtaining a trace to the point where a flaw is found. Because finding attacks on protocols is the central goal to our work, we focus on generating the whole state space and optimising its size through manipulations on the mCRL2 specification and the LPS.

4.3.4 CONFLUENCE

An effective technique for reducing the size of a state space that is generated from an LPS is hiding certain behaviour that we are not interested in and prioritising *confluent* τ actions. This is behaviour that typically occurs when it does not matter in which order certain (independently executing) processes have their actions interleaved. It has been shown that if this is the case and some of these actions are the internal action τ, these can always be chosen before other actions, potentially reducing the size of the state space tremendously. For an in-depth discussion of confluence, please see [**12**].

Using the `lpsconfcheck` tool, those τ actions that are confluent can be marked as such. Then, the LTS generator will prioritise these actions on-the-fly, which may result in a much shorter running time of the LTS generator. On the downside, this only works if there are hidden actions to begin with. In our case that usually means that we must hide certain communicated messages, which may in fact be part of an attack. This means that there may be a trade-off between efficiency and completeness of the attack trace.

Chapter 5 CONVERTING TYPED LYSA TO MCRL2

In this chapter we present the central outcome of our research: a conversion from Typed LySa to mCRL2. Whereas Typed LySa is a higher-level domain-specific language, mCRL2 has been designed to be very generic and widely applicable. This makes the challenge of converting a protocol description to mCRL2 twofold: first, the behaviour of the agents must be accurately translated to the mCRL2 language. Secondly the mCRL2 specification must be extended to include the behaviour of a Dolev-Yao attacker, and the way all agents and the attacker interact must result in a finite and hopefully small state space. As such, a big part of the challenge of this project is optimising this state space, and changing the conversion rules from Typed LySa to mCRL2 accordingly.

In section 5.1 we introduce the nature of the challenge we face. In section 5.2 the general outline of the mCRL2 specification that we create from an input Typed LySa process is discussed, and we present formal conversion rules for a straightforward conversion, initially without an attacker. This straightforward conversion respects the formal semantics of Typed LySa, and may thus be used, for instance, for simulating Typed LySa processes or possibly for validating properties that have no direct relation to security.

A Dolev-Yao attacker is added in section 5.3, but it will be shown that a process that includes such an attacker has an infinitely large state space, which disallows effective automated analysis. We solve this problem in section 5.4, where by using symbolic values we can model many execution paths in one go, at the cost of a more complicated mCRL2 specification. In section 5.5 we present strategies for further reducing the size of the state space, and finally in section 5.6 we show how to find as many flaws as possible while keeping the size of the scenario minimal.

5.1 MODEL CHECKING A SECURITY PROTOCOL

5.1.1 REQUIREMENTS

Model checking, in its purest sense, solves a decision problem where we wish to discover whether or not a certain system exhibits a certain property. In our case, we do not only want to use model checking for verifying the protocol against some security property, but we also wish to find out what went wrong in case this property does not hold. The most commonly requested result is a so-called *attack*, which is a series of steps that an attacker must undertake to violate some security protocol. With tools such as mCRL2's labelled transition system

generator, this is a rather straightforward task. If we stumble upon a state in which a required property does not hold, we can simply save the actions that we performed to end up in this state. This list of actions is called the *trace*, and it is an instance of an attack.

The disadvantage of this wish to reproduce the attack, however, is that many commonly used methods to reduce the size of the state space to something manageably small cannot be used. The central idea of these methods is that we are only interested in a specific part of the system's behaviour, say, a certain action, and we consider all other behaviour *internal behaviour*. By hiding this internal behaviour completely, we can in many cases reduce the size of the state space tremendously, without loss of precision. However, in our case the only actions that are performed are send/receive actions, where a message is posted on the network by an agent and received by agent, respectively, along with a custom `FAIL` action, which we allow to occur if and only if a required security property is violated. If we hide the send/receive actions so that only `FAIL` actions are visible, the state space can be made very small and we can indeed check whether the protocol violates the property, but we cannot reconstruct the attack that led to it.

Combining mCRL2 with LySa

In our case, we wish to combine model checking with LySa's static analysis method. If the LySa tool does not report a violation, then we are certain that there is none, so there is no need for model checking. If it does, however, then we are not sure whether there really is a violation, because LySa's analysis over-approximates the protocol's behaviour. Additionally, LySa's analysis result merely provides some hints as to what might be the problem, but cannot reconstruct an attack.

The reason we use model checking, then, is twofold. First of all, we wish to increase our certainty that there is indeed a flaw in the protocol. This can in fact be done by hiding all internal behaviour and searching for a `FAIL` action. Secondly, we wish to reproduce the attack that leads to this flaw. Because the LySa tool already provides considerable intuition to whether a protocol has a flaw or not, we focus on the situation where a trace is required.

5.1.2 State space explosion

Unfortunately, when we directly transform a (Typed) LySa specification to an mCRL2 model, the size of the state space blows up tremendously. In most situations, it becomes infinitely large. This happens because unlike many other communicating systems, we cannot check these protocols in isolation. Instead, we have to include a custom agent which simulates the best possible attacker. The behaviour of such an attacker has been proposed by Dolev and Yao [**10**] and is described in section 2.1.2.

The central notion of the Dolev-Yao attacker is that it sends any message to any agent at any time, that it could generate with the information available to it. Because mCRL2 relies on synchronous communication, the amount of messages actually sent by the attacker is reduced to only those messages that an agent would accept at that specific point. This may still mean that we can send infinitely many messages, however. For instance, imagine an agent B that inputs a message as follows, in Typed LySa notation:

$$(A, B; xK : N).\, P$$

Where $xK : N$ means that xK is a new variable which will contain a name. Assuming that the attacker knows both A and B, it will send just as many messages to agent B as it has names in its *knowledge*. For example, if the attacker knows the values A, B, N_A, N_B, it can perform these four output operations:

$$\langle A, B, A \rangle . Q$$
$$\langle A, B, B \rangle . Q$$
$$\langle A, B, N_A \rangle . Q$$
$$\langle A, B, N_B \rangle . Q$$

As the attacker's knowledge grows by intercepting more messages on the network, the size this list increases. The attacker can send all of those messages at nearly any moment, and we want to explore every of those options. As such, the state space grows exponentially in the amount of steps of the protocol.

Worse yet, consider the situation if the value xK had been a ciphertext:

$$(A, B; xK : C) . P$$

Now, the attacker can send, for instance, the following messages:

$$\langle A, B, \{N_A\}_{N_A} \rangle . Q$$
$$\langle A, B, \{\{N_A\}_{N_A}\}_{N_A} \rangle . Q$$
$$\langle A, B, \{\{\{N_A\}_{N_A}\}_{N_A}\}_{N_A} \rangle . Q$$
$$etc..$$

Clearly, this yields an infinite state space. In this chapter, we apply several steps to remedy this situation.

5.2 STRAIGHTFORWARD CONVERSION

In this section, we describe a straightforward conversion of a Typed LySa process description to an mCRL2 specification. In the previous section, we showed that in the presence of a Dolev-Yao attacker, this yields an infinite state space. We discuss the implementation of such an attacker nevertheless, for the sake of completion. In later sections we will expand this conversion to gradually improve the results.

5.2.1 GLOBAL STRUCTURE

We initially mimic the Typed LySa semantics as closely as possible, so that it may be easily seen that our conversion is correct. This way, by gradually expanding the conversion to more complex scenarios, the reader may be more easily convinced that also these scenarios are correct. In Appendix C, a proof of the conversion of a minimal variant of Typed LySa to mCRL2 is given.

The next sections describe formal conversion rules by which a Typed LySa process may be converted to an mCRL2 process. However, we first describe the global structure of our mCRL2 output process.

THE META-LEVEL

We will not unfold Typed LySa's meta-level processes to object-level processes and then create an mCRL2 process. Even though this would probably work very well, the resulting mCRL2 process description contains a lot of redundancy and is therefore difficult to read or manipulate. As we will see, improvements and optimisations can often by made by directly manipulating an mCRL2 process instead of the Typed LySa input. As such, we wish the mCRL2 definition to be as simple and well-readable as possible.

Instead of unfolding the meta-level, we use mCRL2's support for variables and process arguments for supporting meta-variables in-place. This means that for every Typed LySa subprocess that is preceded by an indexed parallel operator, we create a new named mCRL2 process, with the meta-variables that it defines, plus any previously defined (meta-)variables, as arguments. This way, Typed LySa meta-variables directly map to mCRL2 variables of type *Nat*.

COMMUNICATION

Corresponding to Typed LySa's semantics, we use synchronous communication. Additionally, we do not wish to model channels of any sort; there is only one channel, the ether. This means that the communication of messages can be very simply modelled by creating two actions `send` and `recv` and making them synchronise to an action `c` in the standard way. We use mCRL2's *List* construct for easily allowing value tuples of any length to be communicated. Synchronous communication is enforced by applying the allow and communication operators to the resulting mCRL2 process P' as follows:

act `recv, send, c:` *List*(`Value`);
`FAIL: Value × Value × Anno;`
init $\nabla_{\{\texttt{c, FAIL}\}}(\Gamma_{\{\texttt{recv}\mid\texttt{send}\rightarrow\texttt{c}\}}P')$;

The `FAIL` action occurs when annotations are violated and will be described in more detail shortly.

DATA TYPES

As described in Chapter 3, Typed LySa knows two distinct types, names and ciphertexts. We model names by creating a structured sort `Name` that has constructors for every name with data elements for every meta-value they may be indexed by. For example, the `Name` sort for our Wide Mouthed Frog example on page 26 will look like this:

sort `Name` = **struct** $A(Nat) \mid B(Nat) \mid S \mid K(Nat, Nat) \mid m(Nat, Nat)$;

The sort of cryptopoints, `CP`, which also can be indexed by meta-values, is defined in a similar way:

sort `CP` = **struct** $a1(Nat, Nat) \mid a3(Nat, Nat) \mid b2(Nat, Nat) \mid b3(Nat, Nat) \mid$
$s1(Nat,Nat) \mid s2(Nat,Nat)$;

Ciphertexts consist of a list of values plus the key that was used to encrypt them. Additionally, we store the annotations provided (if any) directly in the ciphertext, so that we can check their validity when the ciphertext is being decrypted. Now, notice that the key of a ciphertext is simply another value that is pattern-matched against; just like any value occurring before the

semicolon in a Typed LySa `decrypt` operation.[2] We take advantage of this by storing ciphertexts as only a list of values and an annotation, where the first element of the list is in fact the key of the ciphertext:

sort `Ciphertext` = **struct** `Cpair(_vs:` $List$`(Value), _a: Anno);`

This makes especially the implementation of the symbolic attacker, as described in section 5.4, significantly simpler.

In order to be able to communicate both names and ciphertexts, we wrap them into a structured type `Value` that maps to either a name or a ciphertext:

sort `Value` = **struct** `N(_n: Name)? is_N | C(_c: Ciphertext)? is_C;`

Notice that we can use the `is_N` or `is_C` recogniser functions to determine the actual type that the value represents. Annotations are defined as a simple structured type:

sort `Anno` = **struct** `at(_cp: CP, _do:` $List$`(CP)) | at_s(_cp_s: CP);`

Typical annotations are specified using the `at` constructor, where do represents the `dest` or `orig` field. Annotations without such a field have the implicit meaning that any destination/origin cryptopoint is valid, and are specified with the `at_s` constructor. We add two special cryptopoints constructors to the `CP` sort: `CPDY`, which is to be used by the attacker and `UCP`, the unspecified cryptopoint, which is silently used when no annotation is present. Then, an unspecified annotation is written as `at_s(UCP)`.

Finally, we provide a convenience wrapper `encrypt` that creates a `Value` containing a ciphertext from a $List$ of values, a key and (optionally) an annotation to hide the design choice of how we store ciphertexts, so that bugs may be prevented and forward compatibility is safeguarded more easily. Similarly, we define a convenience function `set_anno` that updates the annotation of a given ciphertext value.

map `encrypt:` $List$`(CP) × Value × Anno → Value;`
`encrypt:` $List$`(CP) × Value → Value;`
`set_anno: Value × Anno → Value;`
var k`: Value;`
vs`:` $List$`(Value);`
ad, a, a'`: Anno;`
eqn `encrypt(`vs, k, ad`) = C(Cpair(`$k \rhd vs, ad$`));`
`encrypt(`vs, k, ad`) = C(Cpair(`$k \rhd vs$`, at_s(UCP)));`
`set_anno(C(Cpair(`vs, a`)),` a'`) = C(Cpair(`vs, a'`));`

[2] This only holds because in this project we only consider symmetric cryptography; the situation would be more complex if asymmetric keys are also taken into account.

DECRYPTING CIPHERTEXTS

When a ciphertext is decrypted, we must check two things. First of all, if the pattern matching requirement does not hold, we silently block, as per the semantics of Typed LySa. Secondly, we wish to implement the reference monitor semantics as specified in section 3.3.4: when the annotations specified when the ciphertext was created and those specified upon decryption do not match, the protocol is flawed. In this case, we wish to fire the special action `FAIL` so that it can be found when the state space is generated.

Two annotations match if and only if their cryptopoints are listed in their respective destination/origin sets. We define a mapping `matchDO` which computes exactly that result, taking into account the special annotation `at_s` which means that any cryptopoint is a valid match:

map $\texttt{matchDO}: \texttt{Anno} \times \texttt{Anno} \rightarrow \mathbb{B}$;
var $d, o: \texttt{CP}$;
$D, O: List(\texttt{CP})$;
eqn $\texttt{matchDO}(\texttt{at}(d, D), \texttt{at}(o, O)) = (d \in O) \wedge (o \in D)$;
$\texttt{matchDO}(\texttt{at}(d, D), \texttt{at\_s}(o)) = (o \in D)$;
$\texttt{matchDO}(\texttt{at\_s}(d), \texttt{at}(o, O)) = (d \in O)$;
$\texttt{matchDO}(\texttt{at\_s}(d), \texttt{at\_s}(o)) = true$;

To decrypt a ciphertext that contains new information, we need to create new variables by means of a summation and check that the pattern matching requirement is met. The type-checking part of the pattern matching requirement is directly expressed in the process description as specified in section 5.2.3. Now we can define an auxiliary sub-process `decrypt` that blocks if the pattern matching requirement is not met, issues a `FAIL` if the annotations are violated and silently continues (through the empty multi-action τ) if everything is all right:

proc $\texttt{decrypt}(x: \texttt{Value},\ p: \texttt{Value},\ ao: \texttt{Anno}) =$
$\big(\texttt{is\_C}(x) \wedge \texttt{\_vs}(\texttt{\_c}(x)) \approx \texttt{\_vs}(\texttt{\_c}(p))\big) \rightarrow$
$(\texttt{matchDO}(\texttt{\_a}(\texttt{\_c}(x)), ao) \rightarrow \tau \diamond \texttt{FAIL}(x, p, ao).\delta)$

Notice how `decrypt` and `matchDO` closely correspond to the [DecrRM] rule of Typed LySa's reference monitor semantics (see section 3.3.4). The only difference is that we distinguish between "normal" blocking and blocking when annotations are violated; in the latter case the `FAIL` action occurs before the agent blocks, because that is the situation we are particularly interested in.

As an example of the use of this process, consider a Typed LySa decryption such as

$$\texttt{decrypt}\ y1\ \texttt{as}\ \{A; yK: N\}_{KB}\ [\texttt{at}\ b2\ \texttt{orig}\ \{s2\}]\ \texttt{in}\ P$$

This expression is then, if P' is the mCRL2 conversion of P, typically converted to mCRL2 as follows:

$$\sum_{yK: \texttt{Value}} \texttt{is\_N}(yK) \rightarrow \texttt{decrypt}(y1, \texttt{encrypt}([A,\ yK], KB), \texttt{at}(b2, [s2])).P'$$

In the rest of this section, the rules by which conversions such as these can be done are formalised. The sorts and functions specified in the current section, except for the Name and

CP sorts, do not depend on the Typed LySa input protocol. These are therefore collected in a partial mCRL2 specification that we will call the *preamble* from this point onwards.

5.2.2 CONVERTING DATA EXPRESSIONS

Rules about values and variables will be of the following form:

$$\frac{\Phi \vdash E' \rhd V' \qquad \Phi \vdash E'' \rhd V'' \qquad \cdots}{\Phi \vdash E \rhd V}$$

Here, each E is a Typed LySa data expression, annotation or cryptopoint and each V is an mCRL2 data expression. Φ is a mapping from Typed LySa variable names x to mCRL2 variable names X, which is populated by the rules on processes in the next section.

This notation roughly follows the notation of inference rules and operational semantics. Above the line we find what we have, and below the line what we make of it. What is left of the triangle is Typed LySa code, and right of the triangle an mCRL2 expression that corresponds to it. All identifiers used can be found in the grammar specified in section 3.6.1 or in its commentary.

(Name)	$\dfrac{}{\Phi \vdash a_{i_0,\ldots,i_{n-1}} \rhd \mathtt{N}(a(i_0,\ldots,i_{n-1}))}$
(Var)	$\dfrac{}{\Phi \vdash x_{i_0,\ldots,i_{n-1}} \rhd \Phi(x)}\ (x \mapsto X \in \Phi)$
(Encr)	$\dfrac{\Phi \vdash E_0 \rhd V_0 \ \ldots\ \Phi \vdash E_{n-1} \rhd V_{n-1} \qquad \Phi \vdash \mathcal{A} \rhd \mathcal{A}'}{\Phi \vdash \{E_1,\ldots,E_{n-1}\}_{E_0}\ \mathcal{A} \rhd \mathtt{encrypt}([V_1,\ldots,V_{n-1}], V_0, \mathcal{A}')}$
(Anno)	$\dfrac{\Phi \vdash c_0 \rhd c_0' \ \ldots\ \Phi \vdash c_m \rhd c_m'}{\begin{array}{l}\Phi \vdash [\mathtt{at}\ c_0\ \mathtt{dest}\ \{c_1,\ldots,c_m\}] \rhd \mathtt{at}(c_0', [c_1',\ldots,c_m']) \\ \Phi \vdash [\mathtt{at}\ c_0\ \mathtt{orig}\ \{c_1,\ldots,c_m\}] \rhd \mathtt{at}(c_0', [c_1',\ldots,c_m'])\end{array}}$ $\dfrac{\Phi \vdash c_0 \rhd c_0'}{\Phi \vdash [\mathtt{at}\ c_0] \rhd \mathtt{at\_s}(c_0')} \qquad \dfrac{}{\Phi \vdash [\,] \rhd \mathtt{at\_s}(\mathtt{UCP})}$
(CP)	$\dfrac{}{\Phi \vdash c_{i_0,\ldots,i_{n-1}} \rhd c(i_0,\ldots,i_{n-1})}$

Table 5.1. The conversion relation $\rhd$ for data expressions and cryptopoints

In (Var), the name of a variable x is looked up in Φ; Φ is not otherwise used. We make sure that identifiers such as a and c are defined in mCRL2 by making them constructors of custom `Name` and `CP` sorts, respectively. This is discussed in more detail in section 5.2.4. mCRL2 variable names X are valid identifiers because the rules on process expressions in section 5.2.3 ensure that they are bound by sum expressions or arguments to named processes. Finally, $\mathcal{A}$ is a Typed LySa annotation and $\mathcal{A}'$ is its mCRL2 conversion.

In the Typed LySa syntax, if an encryption or decryption has no associated annotation, then we simply do not write anything. We write [ ] for this situation in the rule (Anno) so that it is easier to read the rule.

5.2.3 CONVERTING PROCESS EXPRESSIONS

Rules about processes will be of the following form:

$$\frac{\Gamma_1, \Phi_1, \mathcal{N}_1, \Upsilon_1 \vdash P_1 \rhd (\Pi_1, P_1') \qquad \Gamma_2, \Phi_2, \mathcal{N}_2, \Upsilon_2 \vdash P_2 \rhd (\Pi_2, P_2') \qquad \ldots}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}$$

Each P is a process expression in Typed LySa and each P' is an mCRL2 process expression. Π is a set of mCRL2 process definitions, which contain parts of the behaviour of the protocol. The tuples $\Gamma, \Phi, \mathcal{N}, \Upsilon$ denote the context information that we keep track of. All four components are partial mappings, as per Table 5.2:

Γ	Maps set identifiers S to the sets $\mathbb{S}$ they denote, i.e. $S \mapsto \mathbb{S} \in \Gamma$
Φ	Maps LySa variable names to mCRL2 variable names, i.e. $x \mapsto X \in \Phi$
$\mathcal{N}$	Maps names to the set identifiers that they are indexed by, i.e. $a \mapsto (S_0, \ldots, S_{n-1}) \in \mathcal{N}$
Υ	Maps meta-variables to their corresponding set identifiers, i.e. $i \mapsto S \in \Upsilon$

Table 5.2. mappings for transformation rules

For any partial mapping $\mathcal{M}$, we write $\mathcal{M}(m)$, $\mathcal{M}[m \mapsto M]$ and $m \in \mathcal{M}$ for querying, updating and testing for the presence of a key, respectively.

The rules presented below are constructed so that all four mappings contain at least those variables, names, etcetera that have been declared in the current scope. In other words, for a declaration $\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P$, the mapping $\mathcal{N}$ collects the free names in P, Φ collects the free variables in P, Γ collects the free set identifiers in P and Υ collects the free meta-variables in P.

Additionally, we use E and V for any Typed LySa and mCRL2 data expression made with the rules in the previous section. All other identifiers found in the mCRL2 expressions are constructions made in the preamble. We start with the conversion rules for the empty process and the `let` operation in Table 5.3.

(Zero)	$\dfrac{}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash 0 \rhd (\emptyset, \delta)}$
(Let)	$\dfrac{\Gamma[S \mapsto \mathbb{S}'], \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \texttt{let}\ S = \mathbb{S}\ \texttt{in}\ P \rhd (\Pi, P')}\ (\mathbb{S}' \subseteq_{\text{fin}} \mathbb{S})$

Table 5.3. Conversion rules for the empty process and `let`

First of all, (Zero) is trivial. (Let) does not change the resulting process, but maps the set identifier S to a *finite* subset of $\mathbb{S}$ to the context variable Γ. As mentioned above, this construction ensures that the same finite subset is chosen for a certain set $\mathbb{S}$ at any place in the process.

(OPar)	$$\frac{\Gamma,\Phi,\mathcal{N},\Upsilon \vdash P_1 \rhd (\Pi_1, P_1') \qquad \Gamma,\Phi,\mathcal{N},\Upsilon \vdash P_2 \rhd (\Pi_2, P_2')}{\Gamma,\Phi,\mathcal{N},\Upsilon \vdash P_1 \mid P_2 \rhd \begin{pmatrix} \Pi_1 \cup \Pi_2 \cup \begin{Bmatrix} p\_A(i_0:\mathbb{N},\dots,i_{n-1}:\mathbb{N},X_0:\texttt{Value},\dots,X_{m-1}:\texttt{Value}) = P_1', \\ p\_B(i_0:\mathbb{N},\dots,i_{n-1}:\mathbb{N},X_0:\texttt{Value},\dots,X_{m-1}:\texttt{Value}) = P_2' \end{Bmatrix}, \\ p\_A(i_0,\dots,i_{n-1},X_0,\dots,X_{m-1}) \parallel p\_B(i_0,\dots,i_{n-1},X_0,\dots,X_{m-1}) \end{pmatrix}} \begin{pmatrix} \Upsilon = \{i_j \mapsto S_j \mid 0 \le j < n\} \\ \Phi = \{x_j \mapsto X_j \mid 0 \le j < m\} \end{pmatrix}$$
(IPar)	$$\frac{\Gamma,\Phi,\mathcal{N},\{i_0 \mapsto S_0,\dots,i_{n-1} \mapsto S_{n-1}\} \vdash P \rhd (\Pi, P')}{\Gamma,\Phi,\mathcal{N},\{i_0 \mapsto S_0,\dots,i_{k-1} \mapsto S_{k-1}\} \vdash \|_{i_k \in S_k,\dots,i_{n-1} \in S_{n-1}} P \rhd \begin{pmatrix} \Pi \cup \{p\_A(i_0:\mathbb{N},\dots,i_{n-1}:\mathbb{N},X_0:\texttt{Value},\dots,X_{m-1}:\texttt{Value}) = P'\}, \\ \|_{i_k \in \mathbb{S}_k,\dots,i_{n-1} \in \mathbb{S}_{n-1}} p\_A(i_0,\dots,i_{n-1},X_0,\dots,X_{m-1}) \end{pmatrix}} \begin{pmatrix} 0 \le k < n \\ \{i_0,\dots,i_{k-1}\} \cap \{i_k,\dots,i_{n-1}\} = \emptyset \\ \Phi = \{x_j \mapsto X_j \mid 0 \le j < m\} \\ \Gamma = \{S_j \mapsto \mathbb{S}_j \mid 0 \le j < n\} \end{pmatrix}$$

Table 5.4. Conversion rules for parallelism

In Table 5.4, p_A and p_B are to be fresh process names. (Opar) converts the ordinary parallel operator. Even though it is not strictly necessary, we create a new mCRL2 process definition for each operand of the operator for each operand process for readability; in typical cases this will produce exactly one process for every role in the protocol. All currently available meta-variable identifiers and variables are passed to the new processes because they may exist as free variables in P_1' and P_2'.

(Ipar) binds new meta-variables and stores them in Υ. The operand process is turned into a process definition, which is called for all combinations of the currently available and newly bound meta-variables. Additionally, we pass currently available variables like in (Opar). Note that we use $\|_{i_k \in \mathbb{S}_k,\dots,i_{n-1} \in \mathbb{S}_{n-1}}$ as a shorthand notation for creating a parallel process call for each element in $\mathbb{S}_k \times \dots \times \mathbb{S}_{n-1}$.

(New)	$$\frac{\Gamma,\Phi,\mathcal{N}[a \mapsto (S_0,\dots S_{n-1})],\Upsilon \vdash P \rhd (\Pi, P')}{\Gamma,\Phi,\mathcal{N},\Upsilon \vdash \left(\nu_{i_k \in S_k,\dots,i_{n-1} \in S_{n-1}}\, a_{i_0,\dots i_{n-1}}\right) P \rhd (\Pi, P')} \begin{pmatrix} 0 \le k \le n, \\ a \notin \mathcal{N}, \\ \Upsilon = \{i_j \mapsto S_j \mid 0 \le j < k\}, \\ \{i_0,\dots,i_{k-1}\} \cap \{i_k,\dots,i_{n-1}\} = \emptyset \\ \forall_{0 \le j < n} S_j \in \Gamma \end{pmatrix}$$
(Outp)	$$\frac{\Phi \vdash E_0 \rhd V_0 \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Gamma,\Phi,\mathcal{N},\Upsilon \vdash P \rhd (\Pi, P')}{\Gamma,\Phi,\mathcal{N},\Upsilon \vdash \langle E_0,\dots,E_{k-1}\rangle.P \rhd (\Pi, \texttt{send}([V_0,\dots,V_{k-1}]) \cdot P')}$$

Table 5.5. Conversion rules for output and restriction

In Table 5.5, (New) adds a newly declared name to the scope variable $\mathcal{N}$, and does not create any mCRL2 code. This works because of the well-formedness restriction that all names in the

process are alpha-renamed apart. In other words, no two $(\nu_X\ \alpha_Y)$ operations can occur where α corresponds to the same identifier and the same meta-sets. (Outp) converts the sending of messages.

(Inp)	$$\frac{\Phi \vdash E_0 \rhd V_0 \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Phi \vdash x_k \rhd X_k \cdots \Phi \vdash x_{n-1} \rhd X_{n-1} \quad \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash (E_0, \ldots, E_{k-1}; x_k{:}T_k, \ldots, x_{n-1}{:}T_{n-1}).P \rhd \left(\Pi, \sum_{X_k,\ldots,X_{n-1}:\texttt{Value}}(\text{is\_}T_k(X_k) \wedge \ldots \wedge \text{is\_}T_{n-1}(X_{n-1})) \rightarrow \texttt{recv}([V_0, \ldots, V_{k-1}, X_k, \ldots, X_{n-1}]) \cdot P'\right)}$$ $$(0 \leq k \leq n \wedge n > 0)$$
(Decr)	$$\frac{\Phi \vdash E_{-1} \rhd V_{-1} \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Phi \vdash c_0 \rhd c_0' \cdots \Phi \vdash c_m \rhd c_m' \quad \Phi \vdash x_k \rhd X_k \cdots \Phi \vdash x_{n-1} \rhd X_{n-1} \quad \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \texttt{decrypt}\ E_{-1}\ \texttt{as}\ \{E_1, \ldots, E_{k-1}; x_k{:}T_k, \ldots, x_{n-1}{:}T_{n-1}\}_{E_0}[\texttt{at}\ c_0\ \texttt{orig}\ \{c_1, \ldots, c_m\}]\ \texttt{in}\ P \rhd \left(\Pi, \sum_{X_k,\ldots,X_{n-1}:\texttt{Value}}(\text{is\_}T_k(X_k) \wedge \ldots \wedge \text{is\_}T_{n-1}(X_{n-1})) \rightarrow \texttt{decrypt}(V_{-1}, \texttt{encrypt}([V_1, \ldots, V_{k-1}, X_k, \ldots, X_{n-1}], V_0), \texttt{at}(c_0', [c_1', \ldots, c_m'])) \cdot P'\right)}$$ $$(1 \leq k \leq n \wedge n > 1)$$
	$$\text{is\_}T = \begin{cases} \texttt{is\_N}, & T = N \\ \texttt{is\_C}, & T = C \end{cases}$$

Table 5.6. Conversion rules for input and decryption

Decryption and input in Table 5.6 are very similar operations; their differences have been mostly smoothed away by `decrypt` subprocess defined in 5.2.1. In both operations, new variables can be bound, for which we create a summand and a condition ensuring that they have the correct type. `recv` is a plain action: here the synchronous communication with `send` ensures that it can only succeed if values are being sent that correspond to the values expected.

Finally, when no more rules can be processed, we finalise our conversion as in Table 5.7.

(Init)	$$\frac{\emptyset, \emptyset, \emptyset, \emptyset \vdash P \rhd (\Pi, P')}{P \rhd \text{preamble; } \textbf{proc}\ \pi_0; \ldots; \pi_{n-1}; \textbf{init}\ \nabla_{\{\texttt{c, FAIL}\}}(\Gamma_{\{\texttt{recv}\mid\texttt{send}\rightarrow\texttt{c}\}}(P'));}$$ $$(\Pi = \{\pi_0, \ldots, \pi_{n-1}\})$$

Table 5.7. The initialisation conversion rule

5.2.4 Name and crypto-point identifiers

Besides transforming the Typed LySa process itself to mCRL2, we need to extract some information so that we can construct mCRL2 data types that correspond to the identifiers used in the Typed LySa protocol. In order not to clutter the conversion rules, we define these separately.

The names used are defined as the largest union of all $\mathcal{N}$ mappings found throughout the conversion. If this resulting union contains duplicate names, then the input was not well-formed and the conversion fails.

The cryptopoints used shall be collected in a separate partial mapping $\mathcal{C}$, which maps each cryptopoint identifier to their *arity*, a number that specifies the amount of meta-variables the cryptopoint identifier has been indexed by. Because all meta-variables in mCRL2 are represented by natural numbers, this is enough. We never need to iterate over all cryptopoints; we only need to declare them as valid mCRL2 identifiers. We fill $\mathcal{C}$ simply by traversing the process definition and adding an entry for every (CP) rule encountered. Similar to the names, if $\mathcal{C}$ would contain the same cryptopoint identifier mapping to a different amount of meta-variables, then the input was not well-formed and the conversion fails.

5.3 ADDING AN ATTACKER

At this point, we have completely converted a Typed LySa protocol into an mCRL2 specification. However, nearly no flaws that a protocol may exhibit will be found, because we run the protocol in a completely isolated environment. All agents perform as described and there are no malicious or dishonest parties involved. As described in section 2.1.2, we can create an environment that simulates the presence of such malicious and dishonest parties by including one additional agent, the Dolev-Yao attacker.

5.3.1 COMMUNICATION MODEL

The Dolev-Yao attacker can read all messages sent on the network. Additionally, it can block these messages, modify them, and generate completely new ones using any knowledge it may have collected. Notice that the attacker can block sent messages, but can recreate and send that same message later on. In other words, in the presence of an attacker, synchronous communication cannot be guaranteed. This holds because there is no way in which a legitimate agent can tell the difference between a message being sent by a legitimate agent or by the attacker: there are no channels and the origin and destination addresses of a message are encoded as plain text values.

To prevent sending unnecessarily many messages, therefore, we model our specification such that all communication goes via the attacker. When an agent sends a message, we say that it *posts that message onto the ether*. We model this so that the agent performs a (synchronous) communication action with the attacker, who adds all elements of the message to its knowledge (possibly decrypting any ciphertext that it knows the key to). When an agent is ready to receive a message, we say that the agent *reads a message from the ether*. This happens through a similar synchronous communication action with the attacker, such that the attacker is able to send any message it can compose from its knowledge. However, only those that actually conform to the pattern matching requirements of the enabled input operation can take place, as ensured by the (Inp) rule in section 5.2.3.

To implement this scenario, we add two more actions, `recvA` and `sendA`, which are performed when the attacker receives or sends a message, respectively. These are forced to communicate with the `send` and `recv` actions performed by the legitimate agents, effectively creating two

channels: one for sending data to the attacker, and one for reading from it. We thus change our initialisation as follows:

act recv, send, recvA, sendA, s, r: $List(\texttt{Value})$;
FAIL: Anno × Anno;
init $\nabla_{\{s,r,\texttt{FAIL}\}}(\Gamma_{\{\texttt{recvA}\mid\texttt{send}\to\texttt{s},\,\texttt{recv}\mid\texttt{sendA}\to\texttt{r}\}}(P' \parallel \texttt{DY}))$;

Here, DY is the attacker process.

5.3.2 The attacker process

A Dolev-Yao attacker process can, in the straightforward setting, be created in a rather simple way. By the structure of the communication model presented above, we guarantee that the attacker can read all the messages sent on the network. We ensure that the attacker can generate information using *any information available* by keeping track of the attacker's *knowledge*, a collection of values that strictly increases as the attacker reads more messages. This knowledge is stored in a struct as follows:

sort Knowledge = **struct** KN(_ns: $List(\texttt{Name})$, _cs: $List(\texttt{Ciphertext})$);

We define a predicate knows which determines whether a certain value can be constructed from this knowledge. This is the case either if the value itself is in one of the lists of the Knowledge sort, or if the value is a ciphertext of which each member can be constructed from the knowledge. The function is defined as follows:

map knows: Knowledge × Value → $\mathbb{B}$;
knows: Knowledge × $List(\texttt{Value})$ → $\mathbb{B}$;
var kn: Knowledge;
vs: $List(\texttt{Value})$;
v: Value;
c: Ciphertext;
n: Name;
eqn $\texttt{knows}(kn, v \rhd vs) = \texttt{knows}(kn, v) \wedge \texttt{knows}(kn, vs)$;
$\texttt{knows}(kn, []) = true$;
$\texttt{knows}(kn, \texttt{C}(c)) = \big(c \in \texttt{\_cs}(kn)\big) \vee \texttt{knows}\big(kn, \texttt{\_vs}(c)\big)$;
$\texttt{knows}(kn, \texttt{N}(n)) = n \in \texttt{\_ns}(kn)$;

Using this function and a function update_knowledge that we will describe below, we can implement the basic behaviour of the DY process as follows:

proc $\texttt{DY}(kn{:}\texttt{Knowledge}) =$
$\sum_{m:List(Value)} \texttt{recvA}(m) \cdot \texttt{DY}(\texttt{update\_knowledge}(kn,\ m))$
$+\ \sum_{m:List(Value)} \texttt{knows}(kn,\ m) \to \texttt{sendA}(m) \cdot \texttt{DY}(kn)$;

This process accepts any message that may be posted onto the either by an agent by means of the send action, which it will add to the attacker's knowledge. Alternatively, it is ready to send any message that can be built from its knowledge and matches the pattern of an enabled recv action in an agent.

UPDATING THE ATTACKER'S KNOWLEDGE

We implement the `update_knowledge` function in such a way that the knowledge is always minimal in size. This is done by first adding all newly received names and ciphertexts to the knowledge if they are not known yet. Then, we check whether there are any known ciphertexts that can be decrypted using any of the known names or ciphertexts as a key. If so, we decrypt the ciphertext, add all contents to the two lists, and remove the decrypted ciphertext from the list of ciphertexts (as we can recreate it using names in the list of names). We repeat this until a fixed point is reached. Note that this is suboptimal, as we do not use the information about which values are newly added and which were already in the knowledge, but it is simpler and more fail-safe. Please see Appendix D for a full specification of this function.

The process we now described is a full Dolev-Yao attacker as per the assumptions in section 2.1.2.[3] However, if we wish to use this attacker for finding flaws in protocols, we must check annotations. This complicates the `update_knowledge` function, because we wish to raise a `FAIL` action whenever a ciphertext is decrypted that does not match the attacker's annotation. We solve the problem by making `update_knowledge` return two values: the new knowledge of the attacker, and a list of ciphertexts that have been newly decrypted. If any of these ciphertexts has an annotation that does not match the attacker's ciphertext `at_s(CPDY)`, a `FAIL` action is raised. Otherwise, `DY` is recursively called with the new knowledge.

Please see Appendix D for a full specification of the `update_knowledge` function and the DY process that includes annotation checking.

5.3.3 IMPROVING THE ATTACKER

As shown in section 5.1.2, using this attacker leads to an infinite state space, because an unlimited amount of nested ciphertexts can be generated. A fairly simple solution to this problem is analysing the protocol specification and creating an attacker that only sends messages that are of the same length and structure as those sent in legitimate protocol runs. This is sufficiently strong, because any message or ciphertext that has a different length or whose elements have a different type than what is expected, will be blocked by Typed LySa's input and decryption operations.

Experiments show that this works well, but even then the state space is tremendously large. The cause of this is the same as the first problem described in section 5.1.2: the amount of messages the attacker can send at any step becomes exponentially large in the amount of values in the attacker's knowledge. Because we wish to be able to produce a trace to any point where a flaw is revealed, our chances of severely reducing the size of this state space are quite small: the common strategy of hiding certain actions as being uninteresting "internal behaviour" does not suit our goals well, because all actions that we have are essential elements

[3] Except for the third assumption which states that the attacker may be either insiders or outsiders; this is not directly related to the `DY` process and is covered in section 5.6.1.

of such a trace. In the next section, we propose a more effective method of keeping our state space within limits.

5.4 THE SYMBOLIC ATTACKER

In one attempt at fixing the state space explosion problem we introduce of a so-called *symbolic ciphertext*, which is a data structure representing *any ciphertext that could be created from the attacker's knowledge at a certain time*. We change the behaviour of the attacker so that it does not send every literal message for any possible ciphertext. Instead, the attacker sends only a symbolic ciphertext, which represents all of those literal ciphertext values. As a result, the state space becomes finite, because all of the cases described in the last example of section 5.1.2 are contained in this one value.

This requires changes to the semantics of the decrypt operation. Instead of checking whether the to-be-decrypted ciphertext has the correct key and correctly matching values, we need to check if such a ciphertext could be built with the knowledge contained in the symbolic ciphertext.

The state space is now finite, but it can still grow very large. For a small protocol such as the BAN Andrew RPC key refreshment protocol [**6**], which is a protocol with only two roles A and B and three messages, the state space is very large and takes several hours to build on a 2 GHz computer, even with a minimal scenario. For larger scenarios or longer protocols (such as the Otway-Rees protocol [**20**], with roles A and B, a server S and five messages), the state space grows too large to fit inside the memory of the computer we used. We decide that this is unacceptably big if the ultimate goal is a push-button solution, so more optimisations are needed.

5.4.1 SYMBOLIC NAMES AND CIPHERTEXTS

Investigations show that the root cause of the state space explosion is that the attacker still sends many separate messages to any agent reading at least one name into a new variable, as per the first example in section 5.2.1. The amount of messages sent is always k^n with k the amount of names that are read into new variables, and n the amount of names in the attacker's knowledge. For example, if the attacker knows the values A, B, NA, NB, M and some agent B reads the following message:

$$(A, B; xK: N, xM: N).\, P$$

then the attacker can send $5^2 = 25$ messages which all will be accepted by agent B, which yields a very large state space.

However, only very few of those 25 messages will yield any interesting results. In general, we can expect the protocol to block soon after an "incorrect " value has been sent to an agent – unless, of course, we have discovered a true flaw in the protocol. Because of this, it makes sense to try to summarise all those 25 messages into one. For this, we invent a second symbolic type, the *symbolic name*, which represents *any name that could be created from the attacker's knowledge at a certain time*.

Using symbolic names and symbolic ciphertexts, we can ensure that only one message is sent from the attacker to a receiving partner at any point in the protocol. Depending on the size of the scenario, the state space can still grow to millions of states, because any agent can typically perform either a send or receive action at any time, leading always to a new unique state. This means that the state space is still $O(k^{2n})$ states, where k is the amount of parallel processes excluding the attacker, and n is the typical amount of messages transferred in one protocol run. This is the case because at any point, every agent (that has not finished running) is ready to either receive or send exactly one message, each of which are independent actions, and after any of these actions is chosen, a new unique state is entered. The exponent is $2n$ and not n because for each message in the traditional protocol narration, two communications are performed: one where a message is posted onto the ether and one where a message is read from it. In section 5.5 we attempt to improve on this situation.

We represent a symbolic name simply by a list of names. Similarly, we represent a symbolic ciphertext by two lists: a list of names, which the attacker has intercepted or decrypted by using a known key, and a list of ciphertexts which the attacker could not decrypt. We use a shorthand notation to describe specific symbolic values as follows: $\{\ldots \mid \ldots\}^s$ is a symbolic ciphertext containing a list of names and a list of ciphertexts, and $[\ldots]^s$ is a symbolic name. For example, $[A, B, C]^s$ is a symbolic name that may be considered equal to the names A, B and C. Similarly, $\{A, B \mid \{C\}_K, \{D, E\}_{K'}\}^s$is a symbolic ciphertext that may be considered equal to infinitely many ciphertexts including, for example, $\{A\}_B, \{\{B\}_B\}_B, \{C\}_K$ and $\{\{C\}_K, A\}_B$.

Notice that the attacker's knowledge, as described in section 5.3.2, has exactly the same structure as the symbolic ciphertext. This is the reason why we sometimes say that a symbolic value *knows* a certain (set of) non-symbolic value(s). More so because symbolic values are only created by the attacker, an untouched symbolic ciphertext can be seen as the set of values that the attacker knew at the point the symbolic ciphertext was received.

The concept of a symbolic attacker is not new. In [**1**] the idea of a lazy intruder is proposed, and our symbolic attacker is essentially an mCRL2 implementation of this idea.

5.4.2 Implementing symbolic data-types in mCRL2

To store symbolic names and values, we create two additional sorts and expand the `Value` sort to support them:

```
sort SName       = List(Name);
sort SCiphertext = struct SCpair(_ns: List(Name), _cs: List(Ciphertext));

sort Value = struct
  N (_n: Name)? is_N
| C (_c: Ciphertext)? is_C
| SN(_sn: SName)? is_SN
| SC(_sc: SCiphertext)? is_SC
```

Additionally, we create two auxiliary recogniser functions `like_C, like_N : Value` → $\mathbb{B}$ that return *true* if their argument is a (symbolic) ciphertext or (symbolic) name, respectively.

We keep these data structures minimal in size by removing any duplicates and by removing a ciphertext from a symbolic ciphertext's list of ciphertexts when its key is known. Note that the latter optimisation is only generally applicable because in this work we only consider symmetric cryptography. When using asymmetric cryptography, being able to decrypt a ciphertext does imply that the same ciphertext can be recreated, so asymmetric ciphertexts would need to be kept in the list of ciphertexts indefinitely.

The semantics of receiving and decrypting, both of which contain pattern matching, unfortunately become significantly more complicated with this optimisation. Both of these operations essentially do two things. First of all, they check if some input value, be it a message sent on the network or a ciphertext that is being decrypted, matches a certain *pattern* and block if not. We call this requirement the *pattern matching requirement*. Secondly, if the requirement holds then any new variables specified in the pattern are assigned the corresponding values from the input value.

The pattern-matching requirement can be modelled as a function that takes two lists of values and decides whether they match or not. In the non-symbolic setup of section 5.2, this function is mostly hidden inside the way communication works in mCRL2. The only part that we have to specify literally is the typing requirement that every new variable must have the specified type. Inspection of the (Decr) and (Inp) operations on page 47 clearly shows this.

For symbolic values, however, we have to write the pattern matching function ourselves. Instead of simply comparing two lists of values, we have to correctly handle the case in which one or more of these values are symbolic values. For instance, consider the case where the variable that we intend to decrypt is a symbolic ciphertext that is to be matched against a list of (non-symbolic) values. The pattern matching function should now return *true* if and only if the whole pattern can be created from the knowledge in the symbolic ciphertext. This pattern matching function becomes rather complex, however, when we also have to consider a situation where the pattern itself contains symbolic values, for instance when pattern matching against variables that were received earlier. Additionally, we must determine which values to assign to the newly created variables.

THE PATTERN MATCHING FUNCTION

We attempt to tackle this complexity by defining one pattern matching function, `pmatch`, which takes two arguments of type `Value`. Recall that for simplicity we represent ciphertexts simply by a list of values where the first value is the ciphertext's key. This key is pattern matched just like any other value that occurs before the semicolon. As such, the ciphertext type can be used for pattern matching any list of values, no matter whether it is later interpreted as an actual ciphertext or simply as a sequence of values. This means that we can use the same `pmatch` function for input and decryption.

Now, consider a situation as follows:

$$\texttt{decrypt}\ x\ \texttt{as}\ \{y;\}_B\ \texttt{in}\ P$$

Both x and y are variables. Furthermore, assume that the variable x is a symbolic ciphertext equal to $\{A, B, C|\emptyset\}^s$ and y is a symbolic name equal to $[B, C, D]^s$. Note that in essence, this

single decryption models multiple paths of execution: one for each combination of the values that can be made from x and y.

This decryption will clearly succeed, because x contains values that can be used to create some of the ciphertexts represented by $\{y;\}_B$. However, x does not know the name D, which means that in the execution path in which y equals D, the decryption should fail. This is a problem when y is used for pattern matching again later in the protocol, for at such a point, y should not anymore represent the execution path in which it equals D. Clearly, we need to restrict the values of y to those which would in fact pass decryption. The same holds for x, which may be considered equal to $\{A\}_B$, a value that cannot possibly match $\{y;\}_B$ because A is not known by y.

This means that in P, we must restrict the values of these variables such that they only represent those values for which the pattern match could have succeeded. We do this by creating new variables x' and y' and substituting any occurrence of x and y in P by them.

THE POST-CONDITION OF PATTERN MATCHING

Because of this complication, we need to compute new values for all variables every time decryption or input occurs. Thus, the return type of `pmatch` cannot be a Boolean, but must be another `Value` in which all symbolic values represent exactly those values the pattern match could have succeeded for. The computation of these values is not dissimilar to computing intersections, which makes `pmatch` a symmetric function (as long as we disregard annotations, which we do for now). Thus, $\texttt{pmatch}(a,b) \approx \texttt{pmatch}(b,a)$.

For example, consider that in the same situation as above, x instead equals the ciphertext $\{[A,B,C]^s\}_K$ and y remains $[B,C,D]^s$. Thus, after $\texttt{pmatch}(x, \{y;\}_K)$ is called, x' should become $\{[B,C]^s\}_K$ and y' should become $[B,C]^s$. These values are obtained by `pmatch`ing each element in the two ciphertexts in a pairwise manner and storing the values in a new ciphertext. Thus, first the two K's are matched, which are equal so the resulting value is also K. Then, $[A,B,C]^s$ and $[B,C,D]^s$ are matched. Clearly, the names for which the pattern matching of two symbolic names could succeed are precisely those names that occur in both symbolic names. This is the intersection of both lists, $[B,C]$. We define similar behaviour for all combinations of names and symbolic names, and of ciphertexts and symbolic ciphertexts, in Table 5.8. We call the result of `pmatch`ing two values against each other the *post-condition of the pattern match*.

The fact that `pmatch` returns a `Value` instead of a Boolean creates a small problem when the pattern match clearly fails. For instance, what value should be returned if two (non-symbolic) names are matched, and they are not equal? For situations such as these, we create the special value `invalid` as an additional constructor of the `Value` sort. We consider any ciphertext that contains an invalid value to be invalid itself. Additionally, any empty symbolic value is also considered invalid. We create an `is_valid` function that determines whether or not a value is valid, and an `ensure_valid` function that returns `invalid` if and only if its argument value is invalid.

The variable z in a pattern such as $\{y; z\colon N\}_B$ is an entirely new variable. As such, in $\texttt{pmatch}(x,\{y; z\colon N\}_B)$ it matches whichever name that there might be in x at that position. For these situations, where we need to specify that any value of the correct type is a correct match, we introduce two more values, `any_name` and `any_ciphertext`, which mean exactly that. This

way, we can treat entirely new variables such as z the same as already existing variables that need updating such as y'. How we ensure that y' and z get assigned the correct values is illustrated in the next subsection.

Now, we can say that a pattern match is successful if and only if `pmatch` returns a valid value. In Table 5.8 we concisely describe how values of any of the underlying types encapsulated by a `Value` are pattern matched with one another. We use N, SN, C and SC as abbreviations for names, symbolic names, ciphertexts and symbolic ciphertexts, respectively. For a precise definition of `pmatch`, please see the mCRL2 definition of the symbolic *preamble* in Appendix D.

Type of a	*Type of* b	*Value of* `pmatch`(a, b) *or* `pmatch`(b, a)
N	N	b if $b \approx a$, `invalid` otherwise.
SN	N	b if $b \in a$, `invalid` otherwise.
SN	SN	A symbolic name containing the intersection of a and b
C	C	A ciphertext containing the `pmatch` result of the pairwise combination of every element in a and b. `invalid` if this list contains `invalid` or if $a \neq b$.
SC	C	A symbolic ciphertext containing all valid results of `pmatch`ing every ciphertext in a to b, plus a ciphertext constructed by `pmatch`ing every element in b to a (if valid).
SC	SC	A symbolic ciphertext containing the intersection of the lists of names in a and b and a list of ciphertexts obtained by `pmatch`ing every ciphertext in a to b and `pmatch`ing every ciphertext in b to a, with all duplicates removed.
`any_name`	N or SN	b.
`any_name`	SC	A symbolic name corresponding to the list of names in b.
`any_ciphertext`	C or SC	b.
Any other combination		`invalid`

Table 5.8. Computing the post-condition of pattern matching symbolic values

In addition to the behaviour specified in Table 5.8, we ensure that `pmatch` always simplifies values in obvious ways: a symbolic ciphertext that contains exactly one ciphertext and no names is rewritten to a (non-symbolic) ciphertext, and a symbolic name that contains exactly one name is rewritten to a name.

INTEGRATING THE PATTERN MATCHING FUNCTION WITH PROCESSES

We illustrate the application of `pmatch` with an example. Consider the following decryption:

$$\texttt{decrypt}\ x\ \texttt{as}\ \{A, B, y; z{:}\ C\}_K\ \texttt{in}\ P$$

Here, A, B and K are names, x and y are existing variables and z is a newly bound variable. Like in the straightforward conversion of section 5.2, decryption is preceded by a sum operation that creates all new variables. In the straightforward conversion, these variables are exactly

the new variables specified in the pattern. With symbolic values, however, we also need to create new versions of the already existing variables that are used for pattern matching.

As a result, in mCRL2 we need to create two versions of the pattern: one with all the values that x must be pattern matched against, and one that contains all new variables in the right location. Then, we can express the pattern matching requirement as the following Boolean expression:

$$\texttt{pmatch}(x, \texttt{encrypt}([A, B, y, \texttt{any\_ciphertext}], K)) \approx \texttt{encrypt}([A, B, y', z], K)$$

Recall that `encrypt` is simply an auxiliary function that creates a ciphertext value from a list of values and a key. If pattern matching fails, the result of the `pmatch` call is `invalid`. This is never equal to the right-hand side of the comparison, so the expression is *false*. If pattern matching does not fail, then this expression is *true* for exactly those values of z and y' that may be a correct post-condition of the pattern match. The special keyword `any_ciphertext` is used to ensure that z can take on any ciphertext that correctly matches with the last element in x (assuming x is non-symbolic; in the other case, z will equal x itself).

Finally, we also need to create a new version of x, the variable that is being decrypted. This variable, x', will need to be exactly equal to the result of the `pmatch` call: it represents exactly those values that could have survived the decryption. Adding the corresponding sum expression, the complete decryption is converted to mCRL2 as follows:

$$\Sigma_{x',y',z:\texttt{Value}} \; x' \approx \texttt{pmatch}(x, \texttt{encrypt}([A, B, y, \texttt{any\_ciphertext}], K)) \wedge x' \approx \texttt{encrypt}([A, B, y', z], K) \rightarrow P$$

To simplify the protocol description, wrap the above Boolean expression in a function `can_pmatch`, which we define for all possible argument types (any valid combination of ciphertexts, symbolic ciphertexts or lists of values). This function is in turn used by the auxiliary processes `decrypt` and `read` that will be defined in section 5.4.3.

Annotations

So far we have disregarded annotations entirely. Because annotations are simply a field of the ciphertext sort, they automatically become part of the input to our pattern matching function. Consider a decryption as follows:

$$\texttt{decrypt}\ x\ \texttt{as}\ \{A, B, y; z{:}\, C\}_K\ [\texttt{at}\ d\ \texttt{orig}\ \{o\}]\ \texttt{in}\ P$$

In the mCRL2 conversion of this expression, $\texttt{pmatch}(x, \{A, B, y; z{:}\, C\}_K)$ will be called, and every y in P will be substituted by a y'. If we assume that x is a normal ciphertext, then it should be clear that we wish to compare the decryption annotation with the annotation attached to x. If x is symbolic, however, this is not so apparent: depending on how x is matched against $\{A, B, y; z{:}\, C\}_K$, the annotation is obtained in a different way. As such, we must match the decryption annotation to the annotation attached to the *result* of our `pmatch` call, and make sure that `pmatch` always retains the correct annotation. We use the following heuristics for determining which annotations to attach to ciphertexts in the result:

1. The resulting ciphertext(s) of the topmost `pmatch` call must always have an annotation derived from the variable x that is being decrypted;
2. Any new variable of type C that is created takes on the annotation from the corresponding ciphertext in x;
3. Any already existing ciphertext (such as y) that is used for pattern matching, must keep its original annotation so that it is unmodified in the post-condition.
4. Symbolic ciphertexts are only created by the attacker, and can only become smaller under subsequent `pmatch` calls. As such, if all values in a ciphertext can be constructed by the values in a symbolic ciphertext, then this describes an execution path in which the whole ciphertext was created by the attacker. In this case, we assign the annotation [`at CPDY`] to the resulting ciphertext.

Depending on the situation, either the annotation from the first or from the second argument to `pmatch` must be retained. Unfortunately, this makes it that `pmatch` is no longer a symmetric function. We tackle this problem by creating two versions of `pmatch` functions that operate on ciphertexts: one that retains annotations in its first argument, and one that retains annotations in its second argument.

THE ATTACKER PROCESS

The symbolic attacker works comparable to the attacker in the straightforward conversion of section 5.3.2: it accepts any message that is posted onto the ether, and appends any new information in it to its knowledge. Every time this happens, the attacker tries to decrypt any ciphertext that it knows with the keys that it knows until no more decryptions can be performed, and checks the annotations of the decrypted ciphertexts, raising `FAIL` upon a violation.

The only significant change to the behaviour of the symbolic attacker process, then, is that in addition to communicating a list of values to an agent that attempts to read from the ether, it sends its complete knowledge. The agent then tries to match the values that it expects to receive to the attacker's knowledge. This way, for example, any ciphertext that is read into a new variable becomes a symbolic ciphertext with the attacker's complete knowledge at that point. Because the communicating action in the agent's specification is preceded by a `pmatch` against the attacker's knowledge, only read actions where the entire pattern is in the attacker's knowledge can occur.

Note that in this case we do not technically need to communicate a list of values at all: just a symbolic ciphertext with the attacker's knowledge suffices. We communicate the list of values anyway, however, because this makes it much more easy to tell different attacker to agent communications apart. This is important, for instance, when running a simulation or reading a trace. As such, the type of the `recv`, `sendA` and `r` actions becomes $List(\texttt{Value}) \times \texttt{SCiphertext}$ instead of just `SCiphertext`.

5.4.3 CHANGES TO THE CONVERSION RULES

Because we need to create new variables for every variable used in pattern matching and translate the pattern twice, the conversion rules for decryption and input need to be slightly modified. The new rules are presented in Table 5.9 and Table 5.10.

These rules use two auxiliary processes, `decrypt` and `read`. The process `decrypt` blocks if the pattern matching requirement fails, performs a τ action if it succeeds, and launches a `FAIL` action if the annotations do not match. A similar process `read` is used for the input operation.

proc $\texttt{decrypt}(x\colon \texttt{Value},\ p\colon \texttt{Value},\ p'\colon \texttt{Value},\ ao\colon \texttt{Anno}) =$
$\quad \sum_{ad\,:\,\texttt{Anno}} \texttt{can\_match}(x, p, \texttt{set\_anno}(p', ad)) \rightarrow$
$\quad\quad (\texttt{matchDO}(ad, ao) \rightarrow \tau \diamond \texttt{FAIL}(x,\ \texttt{set\_anno}(p, ad)).\delta)$

proc $\texttt{read}(p\colon List(\texttt{Value}),\ List(\texttt{Value})) =$
$\quad \sum_{sc\_dy\,:\,\texttt{SCiphertext}} \texttt{can\_match}(sc\_dy, p, p') \rightarrow$
$\quad\quad \texttt{recv}(p, sc\_dy);$

Notice how this version of `decrypt` corresponds to the version of the straightforward conversion described in section 5.2.1.

(SInp)
$$\frac{\begin{array}{c}\Phi \vdash E_0 \ \rhd V_0 \cdots \Phi \vdash E_{k-1} \ \rhd V_{k-1} \\ \Phi' \vdash E_0 \ \rhd V_0' \cdots \Phi' \vdash E_{k-1} \ \rhd V_{k-1}' \qquad \Phi' \vdash x_k \rhd X_k \cdots \Phi' \vdash x_{n+l-1} \rhd X_{n+l-1} \\ \Gamma, \Phi', \mathcal{N}, \Upsilon \ \vdash P \ \rhd (\Pi, P')\end{array}}{\begin{array}{c}\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash (E_0, \dots, E_{k-1}; x_k\colon T_k, \dots, x_{n-1}\colon T_{n-1}).\, P \\ \rhd \\ \left(\Pi, \sum_{X_k,\dots,X_{n+l-1}:\texttt{Value}} \texttt{read}([V_0, \dots, V_{k-1}, \text{any\_t}(T_k), \dots, \text{any\_t}(T_{n-1})], \right. \\ \left. [V_0', \dots, V_{k-1}', X_k, \dots, X_{n-1}]) \cdot P'\right) \\ (0 \le k \le n \ \wedge \ n > 0)\end{array}}$$

(SDecr)
$$\frac{\begin{array}{c}\Phi \vdash E_{-1} \ \rhd V_{-1} \cdots \Phi \vdash E_{k-1} \ \rhd V_{k-1} \qquad \Phi \vdash \mathcal{A} \rhd \mathcal{A}' \\ \Phi' \vdash E_{-1} \ \rhd V_{-1}' \cdots \Phi' \vdash E_{k-1} \ \rhd V_{k-1}' \qquad \Phi' \vdash x_k \rhd X_k \cdots \Phi' \vdash x_{n+l-1} \rhd X_{n+l-1} \\ \Gamma, \Phi', \mathcal{N}, \Upsilon \ \vdash P \ \rhd (\Pi, P')\end{array}}{\begin{array}{c}\Gamma, \Phi, \mathcal{N}, \Upsilon \ \vdash \\ \texttt{decrypt}\ E_{-1}\ \texttt{as}\ \{E_1, \dots, E_{k-1}; x_k\colon T_k, \dots, x_{n-1}\colon T_{n-1}\}_{E_0}\ \mathcal{A}\ \texttt{in}\ P \\ \rhd \\ \left(\Pi, \sum_{X_k,\dots,X_{n+l-1}:\texttt{Value}} \texttt{decrypt}(V_{-1}, \texttt{encrypt}([V_1, \dots, V_{k-1}, \text{any\_t}(T_k), \dots, \text{any\_t}(T_{n-1})], V_0), \right. \\ \left. V_0', \texttt{encrypt}([V_1', \dots, V_{k-1}', X_k, \dots, X_{n-1}], V_0), \mathcal{A}') \cdot P'\right) \\ (1 \le k \le n \ \wedge \ n > 1)\end{array}}$$

(*SDef*)
$$\left(\begin{array}{c}\mathcal{V} = \{x | x \ in \ E_{-1} \dots E_{k-1}\} \\ l = |\mathcal{V}| \\ x_n, \dots, x_{n+l-1} = \mathcal{V} \\ \Phi' = \Phi \ \cup \{x_i \mapsto x_i \mid k \le i < n\} \cup \{x_i \mapsto \Phi(x_i)^{"'"} \mid n \le i < n + l\}\end{array}\right)$$

$$\text{any\_t}(T) = \begin{cases}\texttt{any\_name}, & T = N \\ \texttt{any\_ciphertext}, & T = C\end{cases}$$

Table 5.9. Symbolic conversion rules for input and decryption

First of all, in (*SDef*) we define some common definitions shared by both rules. The first three of these definitions are used to state that in $\mathcal{V}$ we collect all l already-existing variables in the specified pattern, and we call them $x_n, \dots, x_{n+l-1}$ for ease of notation. The fourth definition states that we update Φ, the mapping from Typed LySa variable names to mCRL2 variable names: all

newly declared variables are added, and all existing variables used in the specified pattern are have their names appended by a prime.

Like in the straightforward case of section 5.2, decryption and input are very similar. In both cases, we expand the sum expression to contain the l variables that may need updating, in addition to all newly bound variables. Unlike in the straightforward conversion, we do not add conditions on the types of the new variables; their type is instead ensured by means of the `any_name` and `any_ciphertext` keywords. The other difference with the straightforward case is the presence of the read and decrypt processes, which both accept two renderings of the pattern: one with the original variables and `any_` expressions, that the communicated message or decrypted variable is matched against, and one list of constants and variables that new information is assigned to. The variables occurring in the second list are exactly those variables created in the sum expression.

Finally, we slightly update the initialisation rule to correctly perform communication and add a rule that creates an attacker process at the attacker entry-point in Table 5.10.

	$\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \bullet \rhd (\emptyset, \texttt{DYinit}([\text{inst}(a_0, \overline{S_0}), \dots, \text{inst}(a_{k-1}, \overline{S_{k-1}}), n_\bullet]))$
(DY)	$\begin{pmatrix} k = \lvert\mathcal{N}\rvert \\ \{a_i \mapsto \overline{S_i} \mid 0 \le i < k\} = \mathcal{N} \\ \text{inst}(\alpha, (\mathcal{S}_0, \dots, \mathcal{S}_{n-1})) = \{\alpha(i_0, \dots, i_{n-1}) \mid i_0 \in \mathcal{S}_0, \dots, i_{n-1} \in \mathcal{S}_{n-1}\} \end{pmatrix}$
(SInit)	$\dfrac{\emptyset, \emptyset, \emptyset, \emptyset \vdash P \rhd (\Pi, P')}{P \rhd \text{preamble; } \mathbf{proc}\ \pi_0; \dots; \pi_{n-1}; \mathbf{init}\ \nabla_{\{\texttt{s, r, FAIL}\}} \left(\Gamma_{\{\texttt{recvA}\mid\texttt{send}\to\texttt{s,recvA}\mid\texttt{send}\to\texttt{s}\}}(P')\right); \quad (\Pi = \{\pi_0, \dots, \pi_{n-1}\})}$

Table 5.10. Conversion rules for initialisation and instantiation of the attacker

In (DY), we enumerate all possible values every meta-variable of a known name can have and add it in a list to the `DYinit` process, which launches the attacker process `DY`. In (DY) we instantiate the attacker process, which is defined in the preamble. In addition, we add the unique name $n_\bullet$ which is initially only known by the attacker. This name can be used to simulate keys, nonces, etcetera, that no agent has created but that still may be used to forge an attack. Note that this rule is only correct if the • precedes any indexed parallel operator, which is a well-formedness restriction.

The rule (SInit) may only be applied when no other rules apply anymore and turns the tuple (Π, P') into a valid mCRL2 specification. The preamble defines all functions and auxiliary procedures described in this section as well as all names and cryptopoints used in the protocol, as described in section 5.2.4.

5.5 OPTIMISING THE STATE SPACE

The symbolic attacker is a significant optimisation: the amount of actions that can be performed at any point are reduced from $O(k \cdot c \cdot m)$ in the straightforward case (where the

attacker only sends messages of a structure that may be accepted by an agent), to $O(k)$, where k is the amount of parallel legitimate agents, m is the amount of values in the attacker's knowledge and c is the maximum amount of unspecified values in an enabled input operator.

Nevertheless, the state space can still grow very large: as explained in section 5.4.1, it is exponential in the amount of messages sent in the protocol. In this section, we propose some methods to further optimise the state space. Especially in the case of protocols with many messages and protocols where we need many parallel agents in order to detect attackers, this often pays off.

In this section we propose some additional optimisations to reduce the size of the state space. Note that not all these optimisations have been fully implemented: most notably, we provide no tool that can automate them, so we must often manually modify the mCRL2 specifications generated from Typed LySa input.

The first and foremost observation we make is that every order in which actions from various (often independent) parallel agents are interleaved, is uniquely represented in the state space: the graph representing the labelled transition system that we generate is always a tree, so every state has at most one incoming arc. In many cases, it does not matter in which order actions are interleaved, because most agents act fairly independently. In the next two subsections, we try to reduce some of this redundancy while ensuring that no fewer flaws are found.

5.5.1 PRIORITISING SEND ACTIONS

In most states, the enabled transitions are either the communication from an agent to the attacker (a send action) or communication from the attacker to an agent (a receive action). Every of these enabled actions is performed by a different, independently operating agent, and choosing one of them will never disable another agent's enabled action. This holds because only the attacker's knowledge can influence exactly which messages can be sent at which point: a send action can always occur, and every receive action that matches the values in the attacker's knowledge can occur. Because the attacker never forgets values, it can never be the case that an action that is enabled at some point, gets disabled later on by outside influences.

From this we can conclude that we can give priority to send actions. In other words, when both send and receive actions are enabled, we may disregard the receive actions until no more send actions are possible. The receive actions that were enabled earlier will still be enabled, except that possibly larger symbolic values may be communicated. That is a good thing, because the behaviour represented by operating on some symbolic value v is also included in the behaviour represented by some symbolic value v' that includes all values of v. So, by skipping situations where some variable is bound to such a v and only considering situations where it will be bound to v', we omit redundant paths.

IMPLEMENTATION

Sadly, mCRL2 does not provide a hands-on solution for prioritising certain actions; there is no way to express such behaviour in the mCRL2 language. However, the LTS generator of the mCRL2 toolset does provide us with a useful alternative: this tool has an option that prioritises *confluent τ actions*. As concisely discussed in section 4.3.4, these are internal actions that may

always be picked before other actions, without losing bisimilarity. Typically, other tools (such as mCRL2's `lpsconfcheck`) are used to mark certain actions as confluent: the LTS generator cannot see whether the actions really are confluent. We "abuse" this feature and run the LTS generator in such a way that the send action is considered a confluent τ action. This way, send actions are always prioritised.

There is one downside to this approach however, because send actions are automatically renamed to τ. This would be a good thing if we were merely checking for flaws, but we wish to reproduce a trace to a flaw. With this optimisation, all send actions are hidden, and their data parameters are lost. As such, the trace will only show read actions. This may be enough for someone who is familiar with the protocol to reconstruct the attack, but it still means that our secondary goal, automatically finding attacks on security protocols, is not entirely met.

An alternative is manipulating the LPS that is generated by the lineariser. Because an LPS is basically just a long list of alternatively composed actions preceded by conditions and sum expressions, we can take the disjunction of all the conditions preceding a send action and add the negation of this disjunction condition to every condition preceding a receive action, effectively preventing a receive action from occurring if a send action can occur. LPSs can be manipulated using the library functions of the mCRL2 toolset or by converting an LPS back to an mCRL2 specification, editing the mCRL2 specification by hand and linearising it again. Both approaches require considerable effort that has not currently been made.

5.5.2 Semantically equal states

A state in mCRL2 is uniquely identified by the values of all variables: two states in which the same variables exist and they all have the same values, are the same state. When the LTS generator builds the state space, it searches all available states to see if a to-be-created state already exists, and only actually creates a new state if necessary.

The second problem that contributes to redundancy in the state space, then, is the way we store the attacker's knowledge and symbolic values. In these data-structures, the order in which values are stored does not matter; we might have as well implemented them as sets. However, the current implementation of mCRL2 sets cannot always guarantee that two equal sets are indeed equal, as mentioned in section 4.2.1. In addition, because mCRL2 sets are currently implemented as functions, they are difficult to read by humans, which is important for us when we analyse traces.

This creates the situation that the LTS generator may not recognise two states as being equal even though to the semantics of the protocol they are. Imagine a situation as follows:

$$(\nu\ A)\ \langle A\rangle.\ 0\ |\ (\nu\ B)\ \langle B\rangle.\ 0\ |\ \bullet$$

Here, either the leftmost or the middle process may perform the first send action. Assuming the initial knowledge of the attacker is the empty list [], this list will grow to become either $[A, B]$ or $[B, A]$ depending on which action is first sent. Clearly, these are two different states, even though, assuming the protocol does not end here, exactly the same transitions are possible from both states. This means that the whole state space is duplicated from this point onward: once where the attacker's knowledge is $[A, B]$ and once where it is $[B, A]$.

The simplest way to remedy this problem is to wait for mCRL2's *Set* data type to be more powerful. An alternative would be creating an arbitrary ordering on names and ciphertexts and sorting the list at every point, or creating a data structure supporting ordered values. Finally, an attempt could be made to more finely control the prioritisation of send actions such that they always occur in a certain order, i.e. a send action of agent 1 always has precedence of the a send action of agent 2, etcetera. This will make the order in which the attacker's knowledge increases more predictable, and increases the chance for two states to be combined into one. These alternatives all require considerable effort that currently has not been made, so we give preference to the first option.

5.5.3 Hiding the behaviour of agents

If our primary objective is gaining certainty as to whether or not a protocol is secure, we often wish to run many agents in parallel. This way, attacks can be found in which the attacker uses messages from multiple agents running the protocol simultaneously. However, if two agents run in parallel and perform the same role in the same protocol, an attack involving both agents can occur both ways: the attacker can use messages from agent 1 to fool agent 2, or vice versa, and typically both will be contained in the generated state space.

Reproducing one of these attacks is enough. Therefore, we can completely hide the behaviour of all agents except for one of each role. We cannot do this in the mCRL2 specification itself, because we only want to hide the actions of these agents after they have been forced into communication with the attacker. Therefore, we use the `lpsactionrename` tool to conditionally rename certain read and send actions to τ. If the convention that the first element of a list is always the sending party and the second element is always the receiving party is kept, then this condition is easily formulated. Similarly, we rename all `FAIL` actions in which both cryptopoints correspond to a to-be-hidden agent to τ, so that only traces to flaws in non-hidden agents are detected; these are typically the traces that are most informative.

5.5.4 Not generating the whole state space

Finally, the simplest optimisation is based on the observation that we are not, in fact, interested in the labelled transition system that is generated at all: our goal is to determine whether there are flaws, and if so, produce an attack.

This means that we may simply stop the LTS generator once a flaw has been found and a trace saved. The default exploration strategy used by the LTS generator is to perform a breadth-first search, which suits our needs well: the `FAIL` action that occurs after the lowest amount of performed actions is found first. This `FAIL` generally has the best trace, because no unrelated actions, such as other agents engaging in irrelevant protocol runs, are interleaved.

If the protocol exhibits multiple, fundamentally different, flaws, then of course only one will be found. However, as typically a very large amount of `FAIL` actions are found when the entire state space of a protocol is generated, manually inspecting all of them is a considerable amount of effort, which might be better spent trying to fix the flaw in the Typed LySa protocol narration, and then perform another analysis to confirm that the flaw is fixed.

Naturally, when verifying a secure protocol, we need to explore the entire state space to ensure that the protocol exhibits no flaws (in the specified scenario). If we suspect that no

flaws are present, however, a good strategy is to hide all actions except `FAIL` and use confluence to reduce the size of the state space.

5.6 IMPROVING THE RESULTS

We typically try to limit the size of the scenario we specify, as a large amount of parallel agents quickly increases the amount of time needed for linearisation and state space generation. This may result in not finding potential flaws in protocols, if they can only occur in a scenario that we did not verify. In this section we discuss two methods for finding more potential flaws while keeping the state space as small as possible.

5.6.1 DISHONEST AGENTS

So far, we have not at all taken the third Dolev-Yao assumption from section 2.1.2 into account: "*The attacker may be a legitimate protocol participant (an insider) or an external party (an outsider) or a combination of both*". We have, until now, assumed that all agents that we model are honest and do exactly as they're told, and the attacker is an outsider intruding the network. Dishonest agents, then, are insiders who actively try to violate the stated security properties of a protocol, often by abusing the trust given to them by other protocol participants.

Fortunately, it is easy to include the behaviour of dishonest agents into the attacker process: recall that the attacker can send any message at any time, using any knowledge available and compare this to the behaviour of honest agents. Indeed, by the definition of Typed LySa, honest agents can only use received information or newly created names in their communication with other parties. Because the attacker knows an initial name $n_\bullet$ that it can use to simulate any newly created name, the behaviour of any honest agent is in fact included in the behaviour of a Dolev-Yao attacker.

This means that all that is needed for the attacker to play the role of a dishonest agent is for other agents to willingly communicate with the attacker. Recall that we typically use the value of a meta-variable to uniquely identify an agent; for example, A_1 is agent 1 playing role A. By convention, we assign the value 0 to identify the attacker. Then, if we model our protocol scenario in such a way that every instance of an (honest) agent also attempts to communicate with agent 0, then we have automatically added dishonest agents to our scenario. So, if we want to include dishonest agents, the typical structure of a protocol scenario proposed in section 3.4.3 is changed into the following:

$$
\begin{array}{l}
(\nu_{k\in\{0\}}\ KM_k) \\
(\\
\quad (\nu_{k\in X}\ KM_k) \\
\quad (\ |_{i\in X, j\in X\cup\{0\}} PA_{i,j}\ \mid\ |_{j\in X, i\in X\cup\{0\}} PB_{j,i}\ \mid\ |_{i\in X\cup\{0\}, j\in X\cup\{0\}} S_{i,j}\ ) \\
)\ \mid\ \bullet
\end{array}
$$

We specify that the attacker knows the master key KM_0, to be used in its rule as dishonest agent 0. Additionally, we specify that the processes PA and PB communicate with agent 0, but never act as agent 0. Finally, the server S is willing to facilitate communication between all agents, including agent 0.

Unfortunately, this scenario creates a problem with our conversion rules. The conversion rules from Typed LySa to mCRL2 are defined such that it is not allowed to restrict a name with the same identifier twice – even if the domains of the corresponding sets are disjunct. We remedy the situation with the extra rule in Table 5.11.

There is one last problem occurring, however: each time agent 0, played by the attacker, decrypts a message, the annotation [`at CPDY`] is silently specified, and the same holds for encryptions done by agent 0. Because the attacker is typically not an intended recipient for ciphertexts, this raises unexpected violations of the annotations. We fix this problem by adding `CPDY` to the list of destination/origin cryptopoints if and only if at least one of the meta-variables has the value 0. Note that this workaround is implemented in the conversion tool and in the mCRL2 preamble, but will not be described formally.

(RNew)
$$\frac{\Gamma, \Phi, \mathcal{N}[a \mapsto (S_0, \ldots, S_{k-1}, S_k', \ldots, S_{n-1}')], \Upsilon \vdash P \ \triangleright (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \left(\nu_{i_k \in S_k, \ldots, i_{n-1} \in S_{n-1}} a_{i_0, \ldots i_{n-1}}\right) P \ \triangleright (\Pi, P')}$$

$$\left(\begin{array}{c} 0 \leq k \leq n, \\ \Upsilon = \{i_j \mapsto S_j \mid 0 \leq j < k\}, \\ \{i_0, \ldots, i_{k-1}\} \cap \{i_k, \ldots, i_{n-1}\} = \emptyset, \\ \\ |\mathcal{N}(a)| = n \ \vee \ a \notin \mathcal{N}, \\ \forall_{k \leq j < n} \ S_j \cap \mathcal{N}(a)_j = \emptyset, \\ \forall_{k \leq j < n} \ S_j' = S_j \cup \mathcal{N}(a)_j, \\ \forall_{0 \leq j < n} \ S_j \in \Gamma \end{array}\right)$$

Table 5.11. A restriction rule that allows redefinitions over disjunct sets

(RNew) is a slightly modified version of the (New) specified in section 5.2.3, where we do allow redeclaring names. We only allow this, however, if the same amount of meta-variables is specified and the corresponding sets are disjunct from the sets already mapped to. This way, it is possible to declare a name at multiple locations, provided that the indices declared in-line correspond to sets that are disjunct from the corresponding sets specified at an earlier declaration of the same name. For example, this allows declarations of the form

$$(\nu_{x \in \{0\}} \ a_x) \ldots \bullet \ldots (\nu_{x \in \mathbb{N}^+} \ a_x)$$

Thus, we can specify that only a_0 is part of the attacker's knowledge, and not a_i $(i > 0)$.

5.6.2 Sequential protocol runs

Often a larger amount of attacks is found if multiple agents performing the same role are run in parallel with one another. Unfortunately, it is often not feasible to put many agents in parallel: even the lineariser will slow down tremendously on a level of parallelism that is too high.

A typical type of attack that is found only if each role is run more than one is a *replay attack*, where the attacker captures a message sent in an earlier run and uses it to forge a flaw in a later run, typically when some of the same honest agents are involved. The only way in which we can express running the same process multiple times in Typed LySa is by using indexed parallelism. Very often, however, we are only interested in the case when a process is restarted

only after it has performed its last step, and no earlier. For such cases, parallelism is much too strong a construct that results in an unnecessarily large state space.

There is no way to express this behaviour in Typed LySa, as there are no constructs for looping or recursion. The mCRL2 specification that is generated by the conversion, however, is very suitable for making these kinds of changes. This specification typically contains multiple invocations of named processes put in parallel, as produced by the (IPar) rule on page 46. For example, we might encounter a line as follows:

proc $p\_A = p\_B(1,0) \parallel p\_B(1,2) \parallel \ldots;$

Simply by sequentially duplicating each process invocation, we can obtain the desired effect:

proc $p\_A = p\_B(1,0) \cdot p\_B(1,0) \parallel p\_B(1,2) \cdot p\_B(1,2) \parallel \ldots;$

Note that this does require a minor change to our conversion of Typed LySa's 0 keyword, the empty process. By its semantics, it blocks the process, so we convert it to δ, the deadlock state, in mCRL2. This means that the second invocation of the process is never launched. Without any harm, however, we can convert the empty process to a τ, which means that each named process correctly terminates as long as the protocol is run correctly and no flaws are encountered.

Chapter 6 RESULTS

6.1 ATTACKS FOUND IN KNOWN PROTOCOLS

We have tested our approach on a small collection of well-known protocols. In all cases, all known attacks were found even with relatively small scenarios. We summarise the results in this section. In all cases, the following commands were used:

```
lysa2mcrl2 <file>.tlysa <file>.mcrl2
mcrl22lps –D <file>.mcrl2 <file>.lps
lps2lts –aFAIL –t4 –rjittyc <file>.lps <file>.svc
```

In some cases, the LTS generator was interrupted before the complete state space was built, because an attack was already reported. `lps2lts` saves traces to binary files that we convert to human-readable traces with the following command:

```
tracepp <tracefile>.trc <tracefile>.trcpp
```

WIDE MOUTHED FROG

The Wide Mouthed Frog protocol is a short protocol proposed in [**6**]. The variation that we verified is the one served as a running example in this work:

1. $A \rightarrow S: A, \{B, K\}_{K_{AS}}$
2. $S \rightarrow B: \{A, K\}_{K_{BS}}$
3. $A \rightarrow B: \{m\}_K$

This version has a key freshness attack reported with our method as follows:

```
s([N(Kold)])
s([N(A(1)), N(S), N(A(1)), C(Cpair([N(KA(1)), N(B(2)), N(Kold)], at_s(l1)))])
r([N(A(1)), N(S), N(A(1)), any_ciphertext], SCpair([Kold, known_name, A(1), B(2), S],
   [Cpair([N(KA(1)), N(B(2)), N(Kold)], at_s(l1))]))
FAIL(SC(SCpair([Kold, known_name, A(1), B(2), S], [Cpair([N(KA(1)), N(B(2)),
   N(Kold)], at_s(l1))])), C(Cpair([N(KA(1)), N(B(2)), any_name], at_s(l1))), at(s1(1,
   2), [a1(1, 2)]))
```

The attack is found in a scenario with only one instance of each agent, A_1, B_2 and S. The old key K_{old} is leaked initially, which corresponds to the first two messages of the attack.

NEEDHAM-SCHROEDER SYMMETRIC KEY

The Needham-Schroeder symmetric key protocol [**18**] is possibly the most well-known security protocol, and has a key freshness attack first discovered by Denning and Sacco [**9**]. The protocol goes as follows:

1. $A \rightarrow S: A, B, N_A$
2. $S \rightarrow A: \{N_A, B, K, \{K, A\}_{K_B}\}_{K_A}$
3. $A \rightarrow B: \{K, A\}_{K_B}$
4. $B \rightarrow A: \{N_B\}_K$
5. $A \rightarrow B: \{N_B - 1\}_K$
6. $A \rightarrow B: \{m\}_K$

Here, N_A and N_B are *nonces*, random values that are generated only once. Our analysis reports the attack already at the third step:

```
s([N(Kold)])
s([N(S), N(A(1)), C(Cpair([N(KA(1)), N(NAold), N(B(2)), N(Kold), C(Cpair([N(KB(2)),
   N(A(1)), N(Kold)], at_s(olds1)))], at_s(olds2)))])
s([N(A(1)), N(B(2)), C(Cpair([N(KB(2)), N(A(1)), N(Kold)], at_s(olds3)))])
r([N(A(1)), N(B(2)), any_ciphertext], SCpair([Kold, known_name, A(1), B(2), S, dec],
   [Cpair([N(KB(2)), N(A(1)), N(Kold)], at_s(olds3)), Cpair([N(KA(1)), N(NAold),
   N(B(2)), N(Kold), C(Cpair([N(KB(2)), N(A(1)), N(Kold)], at_s(olds1)))],
   at_s(olds2))]))
FAIL(SC(SCpair([Kold, known_name, A(1), B(2), S, dec], [Cpair([N(KB(2)), N(A(1)),
   N(Kold)], at_s(olds3)), Cpair([N(KA(1)), N(NAold), N(B(2)), N(Kold),
   C(Cpair([N(KB(2)), N(A(1)), N(Kold)], at_s(olds1)))], at_s(olds2))])),
   C(Cpair([N(KB(2)), N(A(1)), any_name], at_s(olds3))), at(b3(1, 2), [s3(1, 2)]))
```

The first three `s` actions are where we intentionally leak the old key `Kold` as well as all messages that it would have been transmitted inside of (messages 2 and 3). The `r` action is message 3 of the protocol, the first action for agent B. It accepts this message because the ciphertext `Cpair([N(KB(2)), N(A(1)), N(Kold)]` is in the attacker's knowledge, which corresponds to $\{A_1, K_{old}\}_{K_{B_2}}$.

A full attack where the attacker plays out agent B can be obtained by removing the origin annotations from the decryptions that belong to messages 3 and 4.

YAHALOM

The Yahalom protocol [**6**] is believed to be secure. However, it is the only protocol in which the LySa tool reports a false positive. The protocol works as follows:

1. $A \rightarrow B: A, N_A$
2. $B \rightarrow S: B, \{A, N_A, N_B\}_{K_B}$
3. $S \rightarrow A: \{B, K, N_A, N_B\}_{K_A}, \{A, K\}_{K_B}$
4. $A \rightarrow B: \{A, K\}_{K_B}, \{N_B\}_K$
5. $A \rightarrow B: \{m\}_K$

We tested a scenario with 2 agents A and B, one server, and instances of agents A and B communicating with the attacker as a dishonest agent, and found no attacks.

6.2 CONCLUSION

In this work we have presented an approach for verifying security protocols. This approach has been specifically designed to complement the analysis results provided by the LySa tool, and performs this task well. When a flaw is found using our approach, a trace is produced which closely resembles the steps an attacker must perform to take advantage of the flaw. This trace, the *attack*, is often the most illustrative vulnerability report, which makes it easy for a protocol designer to fix the flaw. By using symbolic values to summarise many execution paths into one, we are able to keep the state space manageably small.

Given a correct choice of annotations, our method finds all known attacks in the protocols we tested. Additionally, our method reported no errors to be present in protocols that are generally assumed to be secure; this adds to our belief that the analysis is correct and that these protocols are indeed secure, under the network security assumptions of Dolev and Yao and the additional assumption that type flaws cannot occur. As a welcome side-effect, the straightforward conversion is very useful for simulating protocols, which comes in handy when one wishes to make sure the protocol specification has no bugs and typos.

The largest shortcoming of our approach is the general inability to deal with large scenarios; if there is a flaw, the attack is typically found pretty early in state space exploration so there is no need to generate the entire state space. If there is no flaw, however, and we wish to obtain certainty that this is the case for a large scenario, we cannot avoid exploring the entire state space, which may become very large despite several optimisations.

There is a trade-off between the quality of the reported attack and the size of the state space. By hiding more actions, we can reduce the state space size, but obtain less information about how an attack could occur. The LySa tool output may help us to quickly determine the minimal scenario in which a flaw may be found, so that we may not need to hide many actions.

Using symbolic values has been an effective, but time-consuming and complex task. In retrospect, it might have been better to focus on other optimisation techniques which, when combined, may produce equally adequate results. Most specifically, there has not been enough time to investigate the optimisation strategies presented in section 5.5 in detail.

6.3 FURTHER WORK

6.3.1 ASYMMETRIC ENCRYPTION

The largest shortcoming of our current implementation is the lack of support for asymmetric encryption. This disallows many common security protocols to be analysed. Adding support for asymmetric encryption should be a rather straightforward task, however, given enough time.

6.3.2 MORE OPTIMISATION

The mCRL2 toolset provides for many tools and techniques to significantly reduce the state space of a protocol. Not all of these techniques have currently been employed to the fullest yet. Most notably, we have not been able to determine whether send and receive actions may be generally confluent when hidden, or if not, in which circumstances they are. Solving this

question may tremendously improve the speed of our model check, especially in cases when all communication is hidden because there is the suspicion that the protocol is secure. These situations may scale up to large scenarios and still be quickly checked.

6.3.3 TYPES

Even though we called our LySa variant "Typed LySa", its focus on types is rather minimal. In most situations where we may safely assume that every element in a message can be told apart, there is also enough redundancy to include more specific information about the type of each element. This allows an agent to distinguish keys, nonces, agent addresses, etcetera, from one another. In a variant of Typed LySa, the author could be allowed to specify the exact type of each name in a protocol. This allows for a significant reduction of the state space size even in a straightforward conversion, and thus may help us explore larger state spaces.

6.3.4 USING LYSA TOOL OUTPUT

Finally, we have made a few attempts to directly use the LySa Tool's output, but these generally resulted in hiding important parts of an attack in favour of a smaller state space. Given more time, however, the LySa Tool could be of use in developing a push-button solution that is fast enough. For example, the LySa Tool could be used to find the smallest scenario in which a flaw may still be found, after which only this scenario could be model checked.

An alternative and possibly very effective approach is to modify the LySa Tool analysis, for instance by collecting additional information from the protocol narration, with the specific goal of improving the model checking speed. For instance, it may be possible to find a (large) over-approximation of which values may have been communicated between which agents in the path to a single annotation violation. All communications and values that are not in this list may then be safely omitted when model checking to find the attack leading to that violation. Thus we implicitly hide or even entirely omit that part of the state space that cannot contain the attack that we are interested in.

6.3.5 INDUSTRIAL PROTOCOLS

Many automated techniques can verify most academic-produced protocols with relative ease, but it is often not possible to express industrial protocols that are currently in use. These protocols typically have (relatively) much overhead and information that is not directly security-related. Additionally, they often use constructs such as timestamps, simple mathematical operations lifetimes which cannot be expressed in many calculi. Because these protocols are larger and more complex, they may have not been analysed by academic tools as thoroughly as smaller academic protocols have, thus the chance for undiscovered flaws is larger.

Combining static analysis and model checking more tightly may be an excellent technique for verifying security properties of such protocols, and this work may serve as a starting point for such a venture.

BIBLIOGRAPHY

[1] D. Basin, S. Mödersheim, and L. Viganò, "An On-The-Fly Model-Checker for security Protocol Analysis," in *In Proceedings of Esorics'03, LNCS 2808*, 2003, pp. 253-270.

[2] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson, "Static validation of security protocols," *Journal of Computer Security*, vol. 62(1), pp. 1-17, 2004.

[3] C. Boyd and A. Mathuria, *Protocols for Authentication and Key Establishment*. Springer, 2003.

[4] M. Buchholtz, "Automated Analysis of Security in Networking Systems," PhD Thesis, IMM, Danmarks Tekniske Universitet, Kgs. Lyngby, 2004.

[5] M. Buchholtz, F. Nielson, and H. Riis Nielson, "A calculus for control flow analysis of security protocols," *International Journal of Information Security*, 2004.

[6] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, 8(1): 18-36, 1990.

[7] J. Clark and J. Jacob, "A survey of authentication protocol literature: Version 1.0," http://www-users.cs.york.ac.uk/~jac/papers/drareviewps.ps, 1997.

[8] H. Comon-Lundh and V. Cortier, "Security properties: Two agents are sufficient," *Lecture Notes in Computer Science*, vol. 50(1-3), pp. 51-71, 2004.

[9] D. Denning and G. Sacco, "Timestamps in key distributed protocols.," in *Communication of the ACM*, 24(8):533-535, 1981.

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, 29(12): 198-208, 1983.

[11] A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," *Information and Computation*, vol. 148, pp. 36-47, 1999.

[12] J. F. Groote and M. P. A. Sellink, "Confluence for process verification," *Theoretical Computer Science*, 170(1/2):47–81, 1996.

[13] J. F. Groote and T. A. C. Willemse, "Parameterised Boolean Equation Systems," *Theoretical*

Computer Science, 343:332-369, 2005.

[14] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. v. Weerdenburg, "The Formal Specification Language mCRL2," in *Proc. Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings 06351*, 2007.

[15] C. A. R. Hoare, *Communicating sequential processes*. Prentice Hall, 1985, vol. 8.

[16] G. Lowe, "A Hierarchy of Authentication Specifications," in *In CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations*, 1997, p. 31.

[17] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol," *Information Processing Letters*, vol. 56, pp. 131-133, 1995.

[18] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers.," in *Communications of the ACM*, 21(12), 1978.

[19] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes," *Information and Computation* , 100 (100): 1-40, 1980.

[20] D. Otway and O. Rees, "Efficient and timely mutual authentication," *ACM Operating Systems Review*, 1987.

[21] G. Plotkin, "A structural approach to operational semantics," Technical Report FN-19, Department of Computer Science (DAIMI), 1981.

Appendix A THE lysa2mcrl2 TOOL

The conversion rules and mCRL2 definitions described in Chapter 5 have been implemented into a tool called `lysa2mcrl2`. This is a simple command-line tool that takes a Typed LySa process as input and produces an mCRL2 specification as output. The tool has been developed in C++ according to the development guidelines of the mCRL2 toolset, so that it may be included in the toolset if that is desired. A series of options are available with which the user may specify conversion details. Additionally, the mCRL2 preamble and the precise result of converting each Typed LySa expression may be customised by creating custom configuration files.

Both the straightforward conversion of section 5.2 and the symbolic conversion of section 5.4 are enabled, and the latter is the default. The current version of the tool can only include an attacker with the symbolic conversion, however.

A.1 SYNTAX AND OPTIONS

Usage:
lysa2mcrl2 [OPTION]... [INFILE [OUTFILE]]

Converts a security protocol specified in Typed LySa in `INFILE` into an mCRL2 specification in `OUTFILE`. If `OUTFILE` is not present, `stdout` is used. If `INFILE` is not present, `stdin` is used.

Options:

-aNUM, --attacker-index=NUM
Assume that the attacker may be a legitimate (but dishonest) agent participating in the protocol, corresponding to meta-level index number `NUM`. The effect of setting this option is that the attacker's crypto-point `CPDY` is added to all `dest/orig` clauses where one or more of the current meta-variables equal `NUM`. This option corresponds to the `attackerIndex` option of the LySa tool.

-fFILENAME, --fmt-file=FILENAME
Use the format strings in `FILENAME` to build mCRL2 expressions. Defaults to `symbolic.fmt` (or `straightforward.fmt` is `-n` is present).

-i[PREFIX], --prefix-idents[=PREFIX]
Prefixes all identifiers found in the Typed LySa process in `INPUT` with an underscore or with `PREFIX` to prevent clashes with mCRL2 keywords or identifiers used in the preamble

-l, --lysa
Converts a Typed LySa process to LySa and not to mCRL2. Makes all other non-standard options illegal.

-n, --no-attacker
Produces an mCRL2 specification without support for a symbolic attacker. This makes the specification significantly simpler and the state space significantly smaller. However, no Dolev-Yao attacker is included so no outside attacks will be found. Changes the default values of `-p` and `-f`.

-pFILENAME, --preamble=FILENAME
Prepend the mCRL2 preamble in `FILENAME`, instead of using the built-in preamble. The preamble contains all mCRL2 code that is not directly dependent on the input protocol, including all data expressions (except the Name and Ciphertext sorts), the attacker process and auxiliary processes. Defaults to preamble.mcrl2 (or preamble_straightforward.mcrl2 if -n is present).

-z[ACTION], --zero-action[=ACTION]
Generates `ACTION` before deadlocking when Typed LySa's empty process (0) is encountered. Defaults to 'zero' when specified without argument. This is a valid action in the supplied preambles.

Standard options:
-q, --quiet
Do not display warning messages.

-v, --verbose
Display short intermediate messages.

-d, --debug
Display detailed intermediate messages. *(not used by* `lysa2mcrl2`*)*

-h, --help
Display help information.

--version
Display version information.

A.2 GETTING AND RUNNING THE TOOL

The `lysa2mcrl2` tool has been programmed in C++ using the Boost libraries and the Flex and Bison parser toolkit. It is expected to compile on any platform with fairly recent GCC support, which has been tested on Linux and Windows. For convenience, a direct web-interface to the tool is provided at `http://e.teeselink.nl/lysa2mcrl2`, which is also the address where the tool source files and a Windows executable can be downloaded.

The tool sources include a C++ parser for Typed LySa as well as LySa, which may be useful for other projects as well.

Appendix B OVERVIEW OF RULES AND TABLES

B.1 TYPED LYSA

$I ::= \{i_0, \dots, i_{n-1}\} \mid \epsilon$
$IDef ::= \{i_0 \in S_0, \dots, i_{n-1} \in S_{n-1}\}$
$CP ::= c_I$
$E ::= a_I \mid x \mid \{E_0, \dots, E_{n-1}\}_{E_n}[\texttt{at}\ CP_0\ \texttt{dest}\ \{CP_1, \dots, CP_m\}]$
$T ::= N \mid C$
$P ::= \texttt{let}\ S = \mathbb{S}\ \texttt{in}\ P$
$\mid \langle E_0, \dots, E_{n-1}\rangle.P$
$\mid (E_0, \dots, E_{k-1}; x_k : T_k, \dots x_{n-1} : T_{n-1}).P$
$\mid \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1};\ x_k : T_k, \dots x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P\ [\texttt{at}\ CP_0\ \texttt{orig}\ \{CP_1, \dots, CP_m\}]$
$\mid (\nu\ a)\ P$
$\mid (\nu_{IDef}\ a_I)\ P$
$\mid |_{IDef} P$
$\mid P_1 \mid P_2$
$\mid \bullet$
$\mid 0$

$(0 \leq k \leq n \wedge 0 < n)$ *must hold in each rule*

The grammar of Typed LySa

$P \equiv P$

$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$

$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$

$$P_1 \equiv P_2 \Rightarrow \begin{cases} \langle E_0, \dots, E_{n-1}\rangle.\, P_1 \equiv \langle E_0, \dots, E_{n-1}\rangle.\, P_2 \\ (E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}).\, P_1 \equiv (E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}).\, P_2 \\ \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P_1 \equiv \\ \texttt{decrypt}\ E\ \texttt{as}\ \{E_0, \dots, E_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{E_n}\ \texttt{in}\ P_2 \\ (\nu\ n)P_1 \equiv (\nu\ n)P_2 \\ P_1 | P_3 \equiv P_2 | P_3 \end{cases}$$

$P_1 | P_2 \equiv P_2 | P_1$

$(P_1 | P_2) | P_3 \equiv P_1 | (P_2 | P_3)$

$P | 0 \equiv P$

$(\nu\ n)0 \equiv 0$

$(\nu\ n_1)(\nu\ n_2)P \equiv (\nu\ n_2)(\nu\ n_1)P$

$(\nu n)(P_1 | P_2) \equiv P_1 | (\nu\ n)P_2$ if $n \notin$ free names(P_1)

$P_1 \overset{\alpha}{\Leftrightarrow} P_2 \Rightarrow P_1 \equiv P_2$

Structural congruence for Typed LySa

[Comm]
$$\langle D_0, \dots, D_{n-1}\rangle.\, P_1 |\ (D_0, \dots, D_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}).\, P_2 \ \rightarrow\ P_1 |\ P_2 \left[x_k \overset{\alpha}{\rightarrow} D_k, \dots, x_{n-1} \overset{\alpha}{\rightarrow} D_{n-1}\right] \quad \left(\forall_{k \le i < n-1} T_i = \text{typeof}(D_i)\right)$$

[Decr]
$$\texttt{decrypt}\ \{D_0, \dots, D_{n-1}\}_{D_n}\ \texttt{as}\ \{D_0, \dots, D_{k-1}; x_k : T_k, \dots, x_{n-1} : T_{n-1}\}_{D_n}\ \texttt{in}\ P \rightarrow P\left[x_k \overset{\alpha}{\rightarrow} D_k, \dots, x_{n-1} \overset{\alpha}{\rightarrow} D_{n-1}\right] \quad \left(\forall_{k \le i < n-1} T_i = \text{typeof}(D_i)\right)$$

[Onew]
$$\frac{P \rightarrow P'}{(\nu\ n)\ P \rightarrow (\nu\ n)\ P'}$$

[OPar]
$$\frac{P_1 \rightarrow P_1'}{P_1 | P_2 \rightarrow P_1' | P_2}$$

[Congr]
$$\frac{P \equiv P' \quad P' \rightarrow P'' \quad P'' \equiv P'''}{P \rightarrow P'''}$$

$$\text{typeof}(a) = N$$
$$\text{typeof}\left(\{E_0, \dots, E_{n-1}\}_{E_n}\right) = C$$

Reduction relation → for Typed LySa

[DecrRM]	$$\frac{\ell \in \mathcal{L}' \ \wedge \ell' \in \mathcal{L}}{\begin{array}{c}\texttt{decrypt}\ \{D_0, \ldots, D_{n-1}\}_{D_n}\ [\texttt{at}\ \ell\ \texttt{dest}\ \mathcal{L}] \\ \texttt{as}\ \{D_0, \ldots, D_{k-1}; x_k : T_k, \ldots, x_{n-1} : T_{n-1}\}_{D_n}\ [\texttt{at}\ \ell'\ \texttt{orig}\ \mathcal{L}']\texttt{in}\ P \\ \rightarrow_{RM}\ P\left[x_k \xrightarrow{\alpha} D_k, \ldots, x_{n-1} \xrightarrow{\alpha} D_{n-1}\right]\end{array}}$$ $$\left(\forall_{k \leq i < n-1} T_i = \text{typeof}(D_i)\right)$$

Reference monitor semantics for Typed LySa

[Let]	$$\frac{\Gamma[X \mapsto S'], \mathcal{N} \vdash L \Rrightarrow P}{\Gamma, \mathcal{N} \vdash \texttt{let}\ X = \mathbb{S}\ \texttt{in}\ L \Rrightarrow P}\ (S' \subseteq_{\text{fin}} \mathbb{S})$$
[INew]	$$\frac{\Gamma, \mathcal{N} \cup \{\alpha_{i_0,\ldots,i_{k-1}}\overline{a_j} | 0 \leq j < m\} \vdash L \Rrightarrow P}{\begin{array}{c}\Gamma, \mathcal{N} \vdash \left(\nu_{i_k \in X_k, \ldots, i_{n-1} \in X_{n-1}}\ \alpha_{i_0,\ldots,i_{n-1}}\right) L \Rrightarrow \\ \left(\nu\ \alpha_{i_0,\ldots,i_{k-1}}\overline{a_0}\right) \cdots \left(\nu\ \alpha_{i_0,\ldots,i_{k-1}}\overline{a_{m-1}}\right) P\end{array}}\ \begin{pmatrix}\{\overline{a_j} | 0 \leq j < m\} = \Gamma(X_k) \times \ldots \times \Gamma(X_{n-1}) \\ (0 \leq k \leq n)\end{pmatrix}$$
[IPar]	$$\frac{\begin{array}{c}\Gamma, \mathcal{N} \vdash L\left[i_h \mapsto a_{0,h} | 0 \leq h < k\right] \Rrightarrow P_0 \\ \vdots \\ \Gamma, \mathcal{N} \vdash L\langle i_h \mapsto a_{m-1,h} | 0 \leq h < k\rangle \Rrightarrow P_{m-1}\end{array}}{\Gamma, \mathcal{N} \vdash |_{i_0 \in X_0, \ldots, i_{k-1} \in X_{k-1}} L \Rrightarrow P}\ \begin{pmatrix}\{\overline{a_j} | 0 \leq j < m\} = \Gamma(X_0) \times \ldots \times \Gamma(X_{k-1}) \\ 0 < k\end{pmatrix}$$
[Onew]	$$\frac{\Gamma, \mathcal{N} \cup \{\alpha\} \vdash L \Rrightarrow P}{\Gamma, \mathcal{N} \vdash (\nu\ \alpha)\ L \Rrightarrow (\nu\ \alpha)\ P}$$
[OPar]	$$\frac{\Gamma, \mathcal{N} \vdash L_1 \Rrightarrow P_1 \qquad \Gamma, \mathcal{N} \vdash L_2 \Rrightarrow P_2}{\Gamma, \mathcal{N} \vdash L_1 | L_2 \Rrightarrow P_1 | P_2}$$
[Proc]	$$\overline{\Gamma, \mathcal{N} \vdash P \Rrightarrow P}$$
[Open]	$$\overline{\Gamma, \mathcal{N} \vdash \bullet \Rrightarrow P'}$$ for an arbitrary P' such that there are no free variables in P' and all free names in P come from $\mathcal{N}$.

Meta-level to object-level instantiation relation, $\Gamma, \mathcal{N} \vdash L \Rrightarrow P$

B.2 FROM TYPED LYSA TO MCRL2

Γ	Maps set identifiers S to the sets $\mathbb{S}$ they denote, i.e. $S \mapsto \mathbb{S} \in \Gamma$
Φ	Maps LySa variable names to mCRL2 variable names, i.e. $x \mapsto X \in \Phi$
$\mathcal{N}$	Maps names to the set identifiers that they are indexed by, i.e. $a \mapsto (S_0, \ldots, S_{n-1}) \in \mathcal{N}$
Υ	Maps meta-variables to their corresponding set identifiers, i.e. $i \mapsto S \in \Upsilon$

Partial mappings for transformation rules

(Name) $$\frac{}{\Phi \vdash a_{i_0,\ldots,i_{n-1}} \rhd \mathtt{N}(a(i_0,\ldots,i_{n-1}))}$$

(Var) $$\frac{}{\Phi \vdash x_{i_0,\ldots,i_{n-1}} \rhd \Phi(x)} \; (x \mapsto X \in \Phi)$$

(Encr) $$\frac{\Phi \vdash E_0 \rhd V_0 \;\ldots\; \Phi \vdash E_{n-1} \rhd V_{n-1} \qquad \Phi \vdash \mathcal{A} \rhd \mathcal{A}'}{\Phi \vdash \{E_1, \ldots, E_{n-1}\}_{E_0}\, \mathcal{A} \rhd \mathtt{encrypt}([V_1, \ldots, V_{n-1}], V_0, \mathcal{A}')}$$

(Anno) $$\frac{\Phi \vdash c_0 \rhd c_0' \;\ldots\; \Phi \vdash c_m \rhd c_m'}{\begin{array}{l}\Phi \vdash [\mathtt{at}\ c_0\ \mathtt{dest}\ \{c_1, \ldots, c_m\}] \rhd \mathtt{at}(c_0', [c_1', \ldots, c_m']) \\ \Phi \vdash [\mathtt{at}\ c_0\ \mathtt{orig}\ \{c_1, \ldots, c_m\}] \rhd \mathtt{at}(c_0', [c_1', \ldots, c_m'])\end{array}}$$

$$\frac{\Phi \vdash c_0 \rhd c_0'}{\Phi \vdash [\mathtt{at}\ c_0] \rhd \mathtt{at\_s}(c_0')} \qquad \frac{}{\Phi \vdash [\,] \rhd \mathtt{at\_s}(\mathtt{UCP})}$$

(CP) $$\frac{}{\Phi \vdash c_{i_0,\ldots,i_{n-1}} \rhd c(i_0,\ldots,i_{n-1})}$$

The conversion relation $\rhd$ for data expressions and cryptopoints

(Zero) $$\frac{}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash 0 \rhd (\emptyset, \delta)}$$

(Let) $$\frac{\Gamma[S \mapsto \mathbb{S}'], \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \mathtt{let}\ S = \mathbb{S}\ \mathtt{in}\ P \rhd (\Pi, P')} \; (\mathbb{S}' \subseteq_{\text{fin}} \mathbb{S})$$

(OPar) $$\frac{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P_1 \rhd (\Pi_1, P_1') \qquad \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P_2 \rhd (\Pi_2, P_2')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P_1 | P_2 \;\rhd\; \left(\begin{array}{c}\Pi_1 \cup \Pi_2 \cup \left\{\begin{array}{l}p\_A(i_0\colon \mathbb{N}, \ldots, i_{n-1}\colon \mathbb{N}, X_0\colon \mathtt{Value}, \ldots, X_{m-1}\colon \mathtt{Value}) = P_1', \\ p\_B(i_0\colon \mathbb{N}, \ldots, i_{n-1}\colon \mathbb{N}, X_0\colon \mathtt{Value}, \ldots, X_{m-1}\colon \mathtt{Value}) = P_2'\end{array}\right\}, \\ p\_A(i_0, \ldots, i_{n-1}, X_0, \ldots, X_{m-1}) \parallel p\_B(i_0, \ldots, i_{n-1}, X_0, \ldots, X_{m-1})\end{array}\right)}$$

$$\left(\begin{array}{l}\Upsilon = \{i_j \mapsto S_j \mid 0 \le j < n\} \\ \Phi = \{x_j \mapsto X_j \mid 0 \le j < m\}\end{array}\right)$$

(IPar)

$$\frac{\Gamma, \Phi, \mathcal{N}, \{i_0 \mapsto S_0, \ldots, i_{n-1} \mapsto S_{n-1}\} \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \{i_0 \mapsto S_0, \ldots, i_{k-1} \mapsto S_{k-1}\} \vdash \|_{i_k \in S_k, \ldots, i_{n-1} \in S_{n-1}} P \rhd \begin{pmatrix} \Pi \cup \{p\_A(i_0 : \mathbb{N}, \ldots, i_{n-1} : \mathbb{N}, X_0 : \texttt{Value}, \ldots, X_{m-1} : \texttt{Value}) = P'\}, \\ \|_{i_k \in \mathbb{S}_k, \ldots, i_{n-1} \in \mathbb{S}_{n-1}} p\_A(i_0, \ldots, i_{n-1}, X_0, \ldots, X_{m-1}) \end{pmatrix}}$$

$$\begin{pmatrix} 0 \le k < n \\ \{i_0, \ldots, i_{k-1}\} \cap \{i_k, \ldots, i_{n-1}\} = \emptyset \\ \Phi = \{x_j \mapsto X_j \,|\, 0 \le j < m\} \\ \Gamma = \{S_j \mapsto \mathbb{S}_j \,|\, 0 \le j < n\} \end{pmatrix}$$

(New)

$$\frac{\Gamma, \Phi, \mathcal{N}[a \mapsto (S_0, \ldots S_{n-1})], \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \left(\nu_{i_k \in S_k, \ldots, i_{n-1} \in S_{n-1}} a_{i_0, \ldots i_{n-1}}\right) P \rhd (\Pi, P')} \begin{pmatrix} 0 \le k \le n, \\ a \notin \mathcal{N}, \\ \Upsilon = \{i_j \mapsto S_j \,|\, 0 \le j < k\}, \\ \{i_0, \ldots, i_{k-1}\} \cap \{i_k, \ldots, i_{n-1}\} = \emptyset \\ \forall_{0 \le j < n} S_j \in \Gamma \end{pmatrix}$$

(Outp)

$$\frac{\Phi \vdash E_0 \rhd V_0 \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \langle E_0, \ldots, E_{k-1} \rangle . P \rhd (\Pi, \texttt{send}([V_0, \ldots, V_{k-1}]) \cdot P')}$$

(Inp)

$$\frac{\Phi \vdash E_0 \rhd V_0 \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Phi \vdash x_k \rhd X_k \cdots \Phi \vdash x_{n-1} \rhd X_{n-1} \qquad \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P')}{\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash (E_0, \ldots, E_{k-1}; x_k : T_k, \ldots, x_{n-1} : T_{n-1}) . P \rhd \left(\Pi, \begin{array}{l} \sum_{X_k, \ldots, X_{n-1} : \texttt{Value}} \left(\text{is\_}T_k(X_k) \wedge \ldots \wedge \text{is\_}T_{n-1}(X_{n-1})\right) \rightarrow \\ \qquad \texttt{recv}([V_0, \ldots, V_{k-1}, X_k, \ldots, X_{n-1}]) \cdot P' \end{array}\right)}$$

$$(0 \le k \le n \wedge n > 0)$$

(Decr)

$$\frac{\begin{array}{c} \Phi \vdash E_{-1} \rhd V_{-1} \cdots \Phi \vdash E_{k-1} \rhd V_{k-1} \qquad \Phi \vdash c_0 \rhd c_0' \cdots \Phi \vdash c_m \rhd c_m' \\ \Phi \vdash x_k \rhd X_k \cdots \Phi \vdash x_{n-1} \rhd X_{n-1} \\ \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash P \rhd (\Pi, P') \end{array}}{\begin{array}{c} \Gamma, \Phi, \mathcal{N}, \Upsilon \vdash \\ \texttt{decrypt}\ E_{-1}\ \texttt{as}\ \{E_1, \ldots, E_{k-1}; x_k : T_k, \ldots, x_{n-1} : T_{n-1}\}_{E_0} [\texttt{at}\ c_0\ \texttt{orig}\ \{c_1, \ldots, c_m\}]\ \texttt{in}\ P \\ \rhd \\ \left(\Pi, \begin{array}{l} \sum_{X_k, \ldots, X_{n-1} : \texttt{Value}} \left(\text{is\_}T_k(X_k) \wedge \ldots \wedge \text{is\_}T_{n-1}(X_{n-1})\right) \rightarrow \\ \quad \texttt{decrypt}(V_{-1}, \texttt{encrypt}([V_1, \ldots, V_{k-1}, X_k, \ldots, X_{n-1}], V_0), \texttt{at}(c_0', [c_1', \ldots, c_m'])) \cdot P' \end{array}\right) \end{array}}$$

$$(1 \le k \le n \wedge n > 1)$$

$$\text{is\_}T = \begin{cases} \texttt{is\_N}, & T = N \\ \texttt{is\_C}, & T = C \end{cases}$$

The conversion relation $\rhd$ for process expressions

(Init)
$$\frac{\emptyset, \emptyset, \emptyset, \emptyset \ \vdash P \rhd (\Pi, P')}{P \ \rhd \text{preamble; } \mathbf{proc}\ \pi_0; \ldots; \pi_{n-1}; \mathbf{init}\ \nabla_{\{\mathtt{c},\ \mathtt{FAIL}\}}(\Gamma_{\{\mathtt{recv}|\mathtt{send}\rightarrow\mathtt{c}\}}(P'));} \quad (\Pi = \{\pi_0, \ldots, \pi_{n-1}\})$$

The initialisation conversion rule

Type of a	*Type of* b	*Value of* $\mathtt{pmatch}(a,b)$ *or* $\mathtt{pmatch}(b,a)$
N	N	b if $b \approx a$, `invalid` otherwise.
SN	N	b if $b \in a$, `invalid` otherwise.
SN	SN	A symbolic name containing the intersection of a and b
C	C	A ciphertext containing the `pmatch` result of the pairwise combination of every element in a and b. `invalid` if this list contains `invalid` or if $a \neq b$.
SC	C	A symbolic ciphertext containing all valid results of `pmatch`ing every ciphertext in a to b, plus a ciphertext constructed by `pmatch`ing every element in b to a (if valid).
SC	SC	A symbolic ciphertext containing the intersection of the lists of names in a and b and a list of ciphertexts obtained by `pmatch`ing every ciphertext in a to b and `pmatch`ing every ciphertext in b to a, with all duplicates removed.
`any_name`	N or SN	b.
`any_name`	SC	A symbolic name corresponding to the list of names in b.
`any_ciphertext`	C or SC	b.
Any other combination		`invalid`

Computing the post-condition of pattern matching symbolic values

(SInp)

$$\frac{\begin{array}{c}\Phi \vdash E_0 \ \rhd V_0 \cdots \Phi \vdash E_{k-1} \ \rhd V_{k-1} \\ \Phi' \vdash E_0 \ \rhd V_0' \cdots \Phi' \vdash E_{k-1} \ \rhd V_{k-1}' \qquad \Phi' \vdash x_k \rhd X_k \cdots \Phi' \vdash x_{n+l-1} \rhd X_{n+l-1} \\ \Gamma, \Phi', \mathcal{N}, \Upsilon \ \vdash P \ \rhd (\Pi, P')\end{array}}{\begin{array}{c}\Gamma, \Phi, \mathcal{N}, \Upsilon \vdash (E_0, \ldots, E_{k-1}; x_k{:}T_k, \ldots, x_{n-1}{:}T_{n-1}).\, P \\ \rhd \\ \left(\Pi, \sum_{X_k,\ldots,X_{n+l-1}:\texttt{Value}} \ \texttt{read}([V_0, \ldots, V_{k-1}, \text{any\_t}(T_k), \ldots, \text{any\_t}(T_{n-1})], \right. \\ \left. [V_0', \ldots, V_{k-1}', X_k, \ldots, X_{n-1}]) \cdot P'\right)\end{array}}$$

$$(0 \le k \le n \ \wedge \ n > 0)$$

(SDecr)

$$\frac{\begin{array}{c}\Phi \vdash E_{-1} \ \rhd V_{-1} \cdots \Phi \vdash E_{k-1} \ \rhd V_{k-1} \qquad \Phi \vdash \mathcal{A} \rhd \mathcal{A}' \\ \Phi' \vdash E_{-1} \ \rhd V_{-1}' \cdots \Phi' \vdash E_{k-1} \ \rhd V_{k-1}' \qquad \Phi' \vdash x_k \rhd X_k \cdots \Phi' \vdash x_{n+l-1} \rhd X_{n+l-1} \\ \Gamma, \Phi', \mathcal{N}, \Upsilon \ \vdash P \ \rhd (\Pi, P')\end{array}}{\begin{array}{c}\Gamma, \Phi, \mathcal{N}, \Upsilon \ \vdash \\ \texttt{decrypt}\ E_{-1}\ \texttt{as}\ \{E_1, \ldots, E_{k-1}; x_k{:}T_k, \ldots, x_{n-1}{:}T_{n-1}\}_{E_0}\ \mathcal{A}\ \texttt{in}\ P \\ \rhd \\ \left(\Pi, \sum_{X_k,\ldots,X_{n+l-1}:\texttt{Value}} \ \texttt{decrypt}(V_{-1}, \texttt{encrypt}([V_1, \ldots, V_{k-1}, \text{any\_t}(T_k), \ldots, \text{any\_t}(T_{n-1})], V_0), \right. \\ \left. V_0', \texttt{encrypt}([V_1', \ldots, V_{k-1}', X_k, \ldots, X_{n-1}], V_0), \mathcal{A}') \cdot P'\right)\end{array}}$$

$$(1 \le k \le n \ \wedge \ n > 1)$$

(*SDef*)

$$\begin{pmatrix} \mathcal{V} = \{x | x \ in \ E_{-1} \ldots E_{k-1}\} \\ l = |\mathcal{V}| \\ x_n, \ldots, x_{n+l-1} = \mathcal{V} \\ \Phi' = \Phi \ \cup \{x_i \mapsto x_i \mid k \le i < n\} \cup \{x_i \mapsto \Phi(x_i)^{\text{"'"}} \mid n \le i < n + l\} \end{pmatrix}$$

$$\text{any\_t}(T) = \begin{cases} \texttt{any\_name}, & T = N \\ \texttt{any\_ciphertext}, & T = C \end{cases}$$

Symbolic conversion rules for input and decryption

(DY)

$$\Gamma, \Phi, \mathcal{N}, \Upsilon \ \vdash \bullet \ \rhd \ (\emptyset, \texttt{DYinit}([\text{inst}(a_0, \overline{S_0}), \ldots, \text{inst}(a_{k-1}, \overline{S_{k-1}}), n_\bullet])$$

$$\begin{pmatrix} k = |\mathcal{N}| \\ \{a_i \mapsto \overline{S_i} | 0 \le i < k\} = \mathcal{N} \\ \text{inst}(\alpha, (\mathcal{S}_0, \ldots, \mathcal{S}_{n-1})) = \{\alpha(i_0, \ldots, i_{n-1}) \mid i_0 \in \mathcal{S}_0, \ldots, i_{n-1} \in \mathcal{S}_{n-1}\} \end{pmatrix}$$

(SInit)

$$\frac{\emptyset, \emptyset, \emptyset, \emptyset \ \vdash P \rhd (\Pi, P')}{P \ \rhd \text{preamble}; \ \textbf{proc}\ \pi_0; \ldots; \pi_{n-1}; \ \textbf{init}\ \nabla_{\{\texttt{s}, \texttt{r}, \texttt{FAIL}\}} \left(\Gamma_{\{\texttt{recvA}|\texttt{send}\rightarrow\texttt{s}, \texttt{recvA}|\texttt{send}\rightarrow\texttt{s}\}}(P')\right);}$$

$$(\Pi = \{\pi_0, \ldots, \pi_{n-1}\})$$

Symbolic conversion rules for initialisation and instantiation of the attacker

Appendix C **PROOFS**

In order for the correctness our conversion to appear plausible, we hereby provide a conversion proof of a minimal variant of Typed LySa, that we call Tiny LySa, to mCRL2. In Tiny LySa (as well as Typed LySa and LySa), the only transition possible is the communication of values. We show that any Tiny LySa process and its conversion to mCRL2 are strongly bisimilar. We hope that this will help to convince the reader that the more complex conversions as given in Chapter 5 may also be correct.

C.1 **TINY LYSA**

For the first step, we propose a very simple version of Typed LySa: one without a meta-level, without an attacker and even without encryption. Its reduction semantics correspond to those of Typed LySa's object-level semantics, with decryption and typing omitted.

[Comm]	$\langle V_0, \dots, V_{n-1}\rangle.P_1 \mid (V_0, \dots, V_{k-1}; x_k, \dots, x_{n-1}).P_2 \rightarrow P_1 \mid P_2\left[x_k \overset{\alpha}{\rightarrow} V_k, \dots, x_{n-1} \overset{\alpha}{\rightarrow} V_{n-1}\right]$	
[New]	$\dfrac{P \rightarrow P'}{(\nu\ n)\ P \rightarrow (\nu\ n)\ P'}$	[Par] $\dfrac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2}$
[Congr]	$\dfrac{P \equiv P' \quad P' \rightarrow P'' \quad P'' \equiv P'''}{P \rightarrow P'''}$	

Table C.1 Reduction relation → for Tiny LySa

Additionally, we define Tiny LySa's structural congruence in Table C.2.

$$P \equiv P$$
$$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$$
$$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$$
$$P_1 \equiv P_2 \Rightarrow \begin{cases} \langle E_1, \dots, E_k \rangle . P_1 \equiv \langle E_1, \dots, E_k \rangle . P_2 \\ (E_1, \dots, E_j ; x_{j+1} : T_{j+1}, \dots, x_k : T_k) . P_1 \equiv (E_1, \dots, E_j ; x_{j+1} : T_{j+1}, \dots, x_k : T_k) . P_2 \\ (\nu\ n) P_1 \equiv (\nu\ n) P_2 \\ P_1 | P_3 \equiv P_2 | P_3 \end{cases}$$
$$P_1 \overset{\alpha}{\Leftrightarrow} P_2 \Rightarrow P_1 \equiv P_2$$
$$P_1 | P_2 \equiv P_2 | P_1$$
$$(P_1 | P_2) | P_3 \equiv P_1 | (P_2 | P_3)$$
$$P | 0 \equiv P$$
$$(\nu\ n) 0 \equiv 0$$
$$(\nu\ n_1)(\nu\ n_2) P \equiv (\nu\ n_2)(\nu\ n_1) P$$
$$(\nu\ n)(P_1 | P_2) \equiv P_1 | (\nu\ n) P_2 \qquad \text{if } n \notin \text{free names}(P_1)$$

Table C.2. Structural congruence ≡ for Tiny LySa

This congruence relation is also equal to the structural congruence of Typed LySa, except that decryption is omitted.

C.2 ANNOTATIONS IN MCRL2

We wish to prove that the conversion from Tiny LySa to mCRL2 is both sound and complete: the mCRL2 version of any process allows all transitions that exist in the original, and no more than that. It turns out that proving the soundness of this conversion is simpler than its completeness, because the most simple conversion relation is not invertible. This produces the problem that an mCRL2 process that has been built using our conversion rules may in fact correspond to multiple syntactically different Tiny LySa processes: in particular, in Tiny LySa the ν operator is used to bind names, but because of our well-formedness restriction that all names are alpha-renamed apart, we can omit this operator in mCRL2.

We similarly omit this operator in our conversion from Typed LySa to mCRL2; in our conversion, all names are globally defined and available, and the information expressed by the ν operators is solely used for determining the initial knowledge of the attacker during the conversion. This means that even though the processes expressed in Typed LySa and in mCRL2 are strongly bisimilar, the conversion is not semantics preserving. Even though it has no useful application, we include the ν operator in Tiny LySa to show how its correctness can be proven

This means that in a conversion from mCRL2 back to Tiny LySa, we could insert arbitrary ν operators at any location. This would require us to prove that any of those resulting processes are in fact semantically equivalent to the original process, which is difficult because we have not derived a large set of axioms to work with in Tiny LySa.

Instead, we circumvent this problem by extending mCRL2 with annotations (not to be confused with LySa annotations). Annotations are pieces of information that we may insert anywhere in an mCRL2 process, which have no operational meaning. Formally, we add the annotation operator Λ to the calculus with the (untimed) semantics in Table C.3.

$$\frac{p \to \checkmark}{\Lambda_A(p) \to \checkmark} \qquad \frac{p \to p'}{\Lambda_A(p) \to \Lambda_A(p')}$$

$$\text{(AN)} \quad \Lambda_A(p) \equiv p$$

Table C.3. Deduction rules and an axiom for the annotation operator extending mCRL2

Clearly, removing all annotations from a process or inserting annotations at random has no effect on the transitions that may fire. As such, any mCRL2 process with annotations is strongly bisimilar to an mCRL2 process with all annotations removed, which means that the axiom AN holds. This is easy to see, but we will not prove this property.

C.3 CONVERSION

We propose conversion rules from Tiny LySa to mCRL2 in Table C.4

(Opar)	$\dfrac{P_1 \rhd P_1' \qquad P_2 \rhd P_2'}{P_1 \mid P_2 \rhd \nabla_{\{c\}}(\Gamma_{\{recv \mid send \to c\}}(P_1' \parallel P_2'))}$
(New)	$\dfrac{P \rhd P'}{(\nu\, n)\, P \rhd \Lambda_{(\nu\, n)}(P')}$ (Zero) $\dfrac{}{0 \rhd \delta}$
(Outp)	$\dfrac{P \rhd P'}{\langle a_0, \ldots, a_{k-1} \rangle . P \rhd \mathtt{send}([a_0, \ldots, a_{k-1}]) \cdot P'}$
(Inp)	$\dfrac{P \rhd P'}{(a_0, \ldots, a_{k-1}; x_k, \ldots, x_{n-1})\, .\, P \rhd \sum_{x_k, \ldots, x_{n-1}:\mathtt{Value}} \mathtt{recv}([a_0, \ldots, a_{k-1}; x_k, \ldots, x_{n-1}]) \cdot P'}$
(Init)	$\dfrac{P \rhd P'}{P \rhd \mathtt{preamble;\, init}\ P';}$ (*if no further rules apply*)

Table C.4. Conversion relation $\rhd$ from Tiny LySa to mCRL2

Additionally, we assume `preamble` to be the following mCRL2 preamble:

```
sort Value = struct a_0|...|a_{n-1};

act send, recv, c: List(Value);
```

Here $a_0, \ldots, a_{n-1}$ are all names occurring in the Tiny LySa process.

Intuitively, it is not hard to see that this conversion is correct: the parallel operator is interleaving in Tiny LySa's reduction semantics (that is no two actions can occur simultaneously, except through the communication rule). In mCRL2, actions in two processes running in parallel may occur simultaneously through *multi-actions*. However, (OPar) renames

any multi-action `send|recv` with the same data parameters to `c` by means of the Γ operator, and blocks any action other than `c` (the ∇ operator). As such, no multi-actions other than the communication of a list of values can occur, so (OPar) is correct.

This also means that input can occur only for those values of $x_k, \ldots, x_{n-1}$ that are equal to the ones sent simultaneously, because only then the data parameters of `send` and `recv` are equal to one another. This means that pattern-matching communication in (Comm) is also correct. Additionally, the (New) rule is correct because of the well-formedness requirement that any name may be bound only once. We use the created annotation in the formal proof. Finally, (Zero) is trivial, for δ is deadlock in mCRL2.

C.4 PROOF OF CONVERSION

In this section we attempt to provide a full formal proof of the conversion rules specified above. As shorthand notation, we write $[\![P]\!]$ for the mCRL2 conversion of the Tiny LySa process P. In other words, $\forall_P\ P \rhd [\![P]\!]$.

As we have seen in the previous section, mCRL2 annotations have no interesting semantic properties at all, and only have syntactical meaning. Using mCRL2 annotations, the conversion rules have been constructed such that the conversion is a bijection. This implies that each Tiny LySa process has one unique (syntactical) representation in mCRL2 with annotations, and that reversing the conversion using that unique representation yields the original process.

We will prove that this representation is indeed equivalent to the original Tiny LySa process. In other words, we must prove that $P \rightarrow^L Q \Leftrightarrow [\![P]\!] \rightarrow^m [\![Q]\!]$ for any Tiny LySa process P, where $\rightarrow^L$ is a semantics reduction in Tiny LySa and $\rightarrow^m$ is a transition in mCRL2.

Codes such as "V6" or "M" used in the proof annotations refer to mCRL2 axioms[**14**]. Furthermore, $\approx$ is data equality and we write $[x \xrightarrow{\alpha} y]$ for capture-avoiding substitution, instead of $[x/y]$ as used in mCRL2-related documentation.

[COMM]
Given

$$\langle \bar{v}, \bar{u} \rangle . P_1^L \mid (\bar{v}; \bar{x}) . P_2^L \rightarrow^L P_1^L \mid P_2^L[\bar{x} \xrightarrow{\alpha} \bar{u}]$$

We must prove that

$$\nabla_{\{c\}}(\Gamma_{\{recv|send \rightarrow c\}}(\mathtt{send}([\bar{v}, \bar{u}]) \cdot P_1 \| \textstyle\sum_{\bar{x}:\mathtt{Value}} \mathtt{recv}([\bar{v}, \bar{x}]) \cdot P_2)) \rightarrow^m \nabla_{\{c\}}(\Gamma_{\{recv|send \rightarrow c\}}(P_1 \| P_2[\bar{x} \xrightarrow{\alpha} \bar{u}]))$$

And vice versa. Here, we write $\bar{\alpha}$ as a shorthand for a list of names $\alpha_0, \ldots, \alpha_{n-1}$:`Value`, $(0 \leq n)$. We assume that $\bar{x} \notin \mathrm{fv}(\mathtt{send}([\bar{v}, \bar{u}]) . P_1)$, which can be enforced by alpha-renaming if necessary.

$\nabla_{\{c\}}(\Gamma_{\{recv|send \rightarrow c\}}(\mathtt{send}([\bar{v}, \bar{u}]) \cdot P_1 \| \sum_{\bar{x}:\mathtt{Value}} \mathtt{recv}([\bar{v}, \bar{x}]) \cdot P_2))$
= {M}

$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}])\cdot P_1 \| l \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2$$
$$+ \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2 \| l\, send([\overline{v},\overline{u}])\cdot P_1$$
$$+\ send([\overline{v},\overline{u}]).P_1 | \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2\ )))$$

= {LM3, LM5, C3, C4, C5, V4, V5, V6 }

$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}])\cdot (P_1 \| \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2\ )))$$
$$+ \sum_{\bar{x}:Value} \nabla_{\{c\}}\ (\Gamma_{\{recv|send\to c\}}(recv([\overline{v},\bar{x}])\cdot (P_2 \| send([\overline{v},\overline{u}])\cdot P_1)))$$
$$+\ \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}])\cdot P_1 \mid \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2))$$

= {C1, C4, V2, V5, A7}

$$\delta$$
$$+ \sum_{\bar{x}:Value} \delta$$
$$+\ \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}])\cdot P_1 \mid \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}])\cdot P_2))$$

= {A6, S6, S8, SUM1, C4, V5}

$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}]) \mid \sum_{\bar{x}:Value} recv([\overline{v},\bar{x}]))) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2))$$

= {C2…C5, V3…V6 }

$$\sum_{\bar{x}:Value} \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}]) \mid recv([\overline{v},\bar{x}]))) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2))$$

= {Cond1, Cond2}

$$\sum_{\bar{x}:Value} \bar{x} \approx \overline{u} \to$$
$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}]) \mid recv([\overline{v},\bar{x}]))) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2))\ \diamond$$
$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(send([\overline{v},\overline{u}]) \mid recv([\overline{v},\bar{x}]))) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2))$$

= {*Case distinction*, C1}

$$\sum_{\bar{x}:Value} \bar{x} \approx \overline{u} \to$$
$$\nabla_{\{c\}}(c([\overline{v},\bar{x}]) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2)))\ \diamond$$
$$\nabla_{\{c\}}(send([\overline{v},\overline{u}]) \mid recv([\overline{v},\bar{x}]) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2)))$$

= {V1, V2, A7}

$$\sum_{\bar{x}:Value} \bar{x} \approx \overline{u} \to c([\overline{v},\bar{x}]) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2))\ \diamond \delta$$

= {*Sum elimination*}

$$c([\overline{v},\overline{u}]) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}((P_1 \| P_2)[\bar{x} \xrightarrow{\alpha} \overline{u}]))$$

= {$\bar{x} \notin fv(P_1)$, $\xrightarrow{\alpha}$ *is capture-avoiding*}

$$c([\overline{v},\overline{u}]) \cdot \nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2[\bar{x} \xrightarrow{\alpha} \overline{u}]))$$

$\to^m$ {*Deduction rule for sequential composition*}

$$\nabla_{\{c\}}(\Gamma_{\{recv|send\to c\}}(P_1 \| P_2[\bar{x} \xrightarrow{\alpha} \overline{u}]))$$

Because only equalities were used, this proves works in both directions.

[PAR]

Given

$$P_1^L \to^L P_1'^L \Rightarrow P_{1\,1}^L | P_2^L \to^L P_1'^L | P_2^L$$

We must prove that

$$P_1 \to^m P_1' \Rightarrow \nabla_{\{c\}}\Gamma_{\{recv|send\to c\}}(P_1 \| P_2) \to^m \nabla_{\{c\}}\Gamma_{\{recv|send\to c\}}(P_1' \| P_2)$$

And vice versa. We use mCRL2's structural operational semantics given in Tables 11 – 26 in [**14**].

$$P_1 \rightarrow^m P_1'$$
$$\Rightarrow \{4^{th} \textit{ rule in Table 18}\}$$
$$P_1 \| P_2 \rightarrow^m P_1' \| P_2$$
$$\Rightarrow \{2^{nd} \textit{ rule in Table 25}\}$$
$$\Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1 \| P_2) \rightarrow^m \Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1' \| P_2)$$
$$\Rightarrow \{2^{nd} \textit{ rule in Table 22}\}$$
$$\nabla_{\{\mathsf{c}\}}\Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1 \| P_2) \rightarrow^m \nabla_{\{\mathsf{c}\}}\Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1' \| P_2)$$

The reverse case is trivial, because the Tiny LySa conversion of

$$P_1 \rightarrow^m P_1' \Rightarrow \nabla_{\{\mathsf{c}\}}\Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1 \| P_2) \rightarrow^m \nabla_{\{\mathsf{c}\}}\Gamma_{\{\mathsf{recv}|\mathsf{send}\rightarrow\mathsf{c}\}}(P_1' \| P_2)$$

corresponds precisely to rule [Par]

[NEW]

Given

$$P^L \rightarrow^L P'^L \Rightarrow (\nu\, n)\, P^L \rightarrow^L (\nu\, n)\, P'^L$$

We must prove that

$$P \rightarrow^m P' \Rightarrow \Lambda_{(\nu\, n)}(P) \rightarrow^m \Lambda_{(\nu\, n)}(P')$$

$$P \rightarrow^m P'$$
$$\Rightarrow \{2^{nd} \textit{ rule in Table C.3}\}$$
$$\Lambda_{(\nu\, n)}(P) \rightarrow^m \Lambda_{(\nu\, n)}(P')$$

The reverse case is trivial, because the Tiny LySa conversion of

$$P \rightarrow^m P' \Rightarrow \Lambda_{(\nu\, n)}(P) \rightarrow^m \Lambda_{(\nu\, n)}(P')$$

corresponds precisely to rule [New]. Note that this would not be the case without the Λ operator.

[CONGR]

To prove this rule, we must prove the correctness of all congruence rules in Table C.2. We discuss the correctness of these more concisely than we have the other rules, because most are trivial. The first 8 rules in Table C.2 directly correspond with rules in Table 2 of [**14**], the derivation rules for mCRL2 processes, which defines equalities not dissimilar to that in Table C.2. The next two rules, which express the associativity and commutativity of the parallel operator, are trivial because the parallel operator of mCRL2 is also associative and commutative, as demonstrated by the 4th rule of Table 18 in [**14**]. Finally, the rules involving the ν operator are correct because their conversion only adds a Λ operator to the mCRL2 conversion, which can be inserted or removed at any point as per the axiom AN.

C.5 MCRL2 WITHOUT ANNOTATIONS

From axiom AN it follows that any mCRL2 process with annotations is strongly bisimilar to the same mCRL2 process with all annotations removed. Because the mCRL2 toolset does not in fact support annotations, we wish to convert our mCRL2 process to one without any annotations in it. We do this by recursively applying axiom AN from left to right.

Wrapping it all up, we have now shown that an mCRL2 process created from some Tiny LySa process, by applying the conversion relation ⊳ and subsequently removing all annotations, is strongly bisimilar to the original Tiny LySa process.

Appendix D mCRL2 PREAMBLES

D.1 PREAMBLE FOR STRAIGHTFORWARD CONVERSION

```
sort Anno = struct at(_cp: CP, _do: List(CP)) | at_s(_cp_s: CP);
map
  AnnoDY: Anno;
  AnnoNone: Anno;
var
  vs: Values;
  a, a': Anno;
eqn
  AnnoDY   = at_s(CPDY);
  AnnoNone = at_s(UCP);

sort Values = List(Value);

sort Ciphertext = struct Cpair(_vs: List(Value), _a: Anno);


sort Value = struct
    N (_n: Name)?is_N
  | C (_c: Ciphertext)?is_C
  ;

map
% more intuitive names for ciphertext elements

  encrypt: Values # Value # Anno -> Value;
  encrypt: Values # Value -> Value;
  encrypt: Values # Anno -> Value;
  encrypt: Values -> Value;
var
  c: Ciphertext;
  v, k: Value;
  vs: Values;
  ad: Anno;
eqn
  encrypt(vs, k, ad) = C(Cpair(k |> vs, ad));
  encrypt(vs, k)     = C(Cpair(k |> vs, AnnoNone));
  encrypt(vs, ad)    = C(Cpair(vs, ad));
  encrypt(vs)        = C(Cpair(vs, AnnoNone));
```

```
map
  matchDO: Anno#Anno -> Bool;
var
  o, d: CP;
  O, D: List(CP);
eqn
  matchDO(at(d, D), at(o, O)) = (d in O) && (o in D);
  matchDO(at_s(d),  at(o, O)) = (d in O);
  matchDO(at(d, D), at_s(o))  = (o in D);
  matchDO(at_s(d),  at_s(o))  = true;

act
  send, recv, c: List(Value);
  FAIL: Value # Value # Anno;
  zero;


proc decrypt(x: Value, p: Value, ao: Anno) =
  ((is_C(x)) && (_vs(_c(x)) == _vs(_c(p)))) ->
  (
    matchDO(_a(_c(x)), ao) ->
      tau
    <>
      FAIL(x, p, ao).delta
  )
  <>
  delta;

% end of preamble.
```

D.2 THE DOLEV-YAO ATTACKER

This functions as an addition to the preamble of section D.2.

```
act
  recvA, sendA: List(Value);

map
  uniq_Value: List(Value) -> List(Value);
  filter_Ciphertext: List(Ciphertext) # (Ciphertext->Bool) -> List(Ciphertext);
  get_Values_from_Ciphertexts: List(Ciphertext) -> List(Value);
var
  v: Value;
  lv: List(Value);
  c: Ciphertext;
  lc: List(Ciphertext);
  f: (Ciphertext->Bool);
eqn
  %uniqueness filter: discards duplicates in list
  uniq_Value(v |> lv) = if(v in lv, uniq_Value(lv), v |> uniq_Value(lv));
  uniq_Value([]) = [];

  %list filter: discareds elements for this f(d) does not hold
  filter_Ciphertext(c |> lc, f) = if(f(c), c |> filter_Ciphertext(lc, f), filter_Ciphertext(lc,
f));
  filter_Ciphertext([], f) = [];

  %auxiliary function that turns a list of ciphertexts into a list of values
  get_Values_from_Ciphertexts(c |> lc) = _vs(c) ++ get_Values_from_Ciphertexts(lc);
  get_Values_from_Ciphertexts([]) = [];


sort Knowledge = struct KN(_ns: List(Name), _cs: List(Ciphertext));
sort KnowledgeUpdate = struct KU(_kn: Knowledge, _dc: List(Ciphertext));
```

```
map
  update_knowledge: Knowledge # List(Value) -> KnowledgeUpdate;
  update_knowledge: KnowledgeUpdate # List(Value) -> KnowledgeUpdate;

  propagate: KnowledgeUpdate -> KnowledgeUpdate;

  cs_can_decrypt: List(Ciphertext) # Knowledge -> List(List(Ciphertext));

var
  ku: KnowledgeUpdate;
  dc: List(Ciphertext);
  kn, kn': Knowledge;
  v, v', k: Value;
  c, c': Ciphertext;
  n, n': Name;
  ns: List(Name);
  cs: List(Ciphertext);
  m: List(Value);
eqn
  update_knowledge(kn, m) = propagate(update_knowledge(KU(kn, []), m));

  update_knowledge(ku, []) = ku;
  update_knowledge(KU(kn, dc), v |> m) =
    update_knowledge(KU(
      if(is_C(v), KN(_ns(kn), _c(v) |> _cs(kn)), KN(_n(v) |> _ns(kn), _cs(kn))),
      dc), m);

%%propagate: KnowledgeUpdate -> KnowledgeUpdate;
  % fixed point recursion: as long as there are undecrypted ciphertexts in the knowledge
  % that can be decrypted, keep doing so.
  %
  % this is suboptimal, because in fact we only need to try to decrypt newly received /
  % decrypted names and ciphertexts. this is much simpler, however.

  propagate(KU(KN(ns, cs), dc)) =
    if(cs_key_known_pair.0 == [],
      KU(KN(ns, cs), dc),
      update_knowledge(
        KU(KN(ns, cs_key_known_pair.1), cs_key_known_pair.0 ++ dc),
        uniq_Value(get_Values_from_Ciphertexts(cs_key_known_pair.0))  %fixme
      )
    )
    whr
      cs_key_known_pair = cs_can_decrypt(cs, KN(ns, cs))
    end;
```

```
%%cs_can_decrypt: List(Ciphertext) # Knowledge -> List(List(Ciphertext));
  % returns a pair of lists of ciphertexts. the first element contains ciphertexts that
  % can be decrypted. the second element contains ciphertexts that cannot be decrypted.
  %
  % loops through the ciphertexts to see if the key can be pmatched to (found in) kn.
  % if so, the pattern matched result replaces the key, and the whole is prepended to the
  % first list in returned list.
  %
  % otherwise, it is appended to the second list.
  %
  % this effectively means that if a ciphertext has a symbolic key which can only be partly
  % matched, the other values represented by the key are discarded. this is allright, because
  % one matched key already suffices for unlocking the other elements of the ciphertext, and
  % an empty ciphertext with only a key has no value to the attacker, because it cannot find
  % that key until it otherwise receives it, somehow.

  cs_can_decrypt(c |> cs, kn) =
    if(knows(kn, C(c)),
      [c |> recurse.0, recurse.1],
      [recurse.0, c |> recurse.1]
    )
    whr
      recurse = cs_can_decrypt(cs, kn)
    end;

  cs_can_decrypt([], kn) = [[],[]];


map
  knows: Knowledge # Value -> Bool;
  knows: Knowledge # List(Value) -> Bool;
var
  kn: Knowledge;
  c: Ciphertext;
  n: Name;
  vs: List(Value);
  v: Value;
eqn
  knows(kn, v |> vs) = knows(kn, v) && knows(kn, vs);
  knows(kn, [])      = true;
  knows(kn, C(c))    = (c in _cs(kn)) || knows(kn, _vs(c));
  knows(kn, N(n))    = n in _ns(kn);


map
  filter_ku_rm: KnowledgeUpdate -> KnowledgeUpdate;
var
  kn: Knowledge;
  cs: List(Ciphertext);
eqn
  filter_ku_rm(KU(kn, cs)) = KU(kn, filter_Ciphertext(cs, lambda c: Ciphertext.!matchDO(_a(c),
AnnoDY)));
```

```
proc DYinit(ns: List(Name)) = DY(KN(ns, []));

proc DY(kn: Knowledge) =
  sum m: List(Value).
    recvA(m).
    %single-point domain sum for not having to compute ku twice.
    sum ku:KnowledgeUpdate.(ku==filter_ku_rm(update_knowledge(kn, m))) ->

      % list was empty? good, the attacker did not decrypt anything it wasn't allowed to.
      (_dc(ku)==[]) ->
        DY(_kn(ku))
      <>
        % otherwise, we raise a FAIL for every not-allowed decryption by the attacker.
        (
          sum c: Ciphertext.(c in _dc(ku)) ->
            FAIL(C(c), C(c), AnnoDY).delta
        )
+
  sum m: List(Value).
    sendA(m).
    DY(kn);
```

D.3 PREAMBLE FOR THE SYMBOLIC ATTACKER

The first collection of mappings define some sort of standard library for the `Value` sort. We define the same functions also for the `Ciphertext` and `Name` sorts, but do not print them here. They can be obtained by replacing every occurrence of `Value` by `Ciphertext` resp. `Name`.

```
map
  forall_Value: List(Value) # (Value->Bool) -> Bool;
  exists_Value: List(Value) # (Value->Bool) -> Bool;
  filter_Value: List(Value) # (Value->Bool) -> List(Value);
  uniq_Value: List(Value) -> List(Value);
  collect2_Value: List(Value) # List(Value) # (Value#Value->Value) -> List(Value);
  collect_Value:  List(Value) # (Value->Value) -> List(Value);
  assert_Value: Bool # Value -> Value;
  FAIL_Value: Value -> Value;
var
  d, d': Value;
  l, l': List(Value);
  ll: List(List(Value));
  f: (Value->Bool);
  ft2: (Value#Value->Value);
  ft: (Value->Value);
  b: Bool;
eqn
  %bounded forall over list elements
  forall_Value(d |> l, f) = f(d) && forall_Value(l, f);
  forall_Value([], f) = true;

  %bounded exists over list elements
  exists_Value(d |> l, f) = f(d) || exists_Value(l, f);
  exists_Value([], f) = false;

  %list filter: discareds elements for this f(d) does not hold
  filter_Value(d |> l, f) = if(f(d), d |> filter_Value(l, f), filter_Value(l, f));
  filter_Value([], f) = [];

  %uniqueness filter: discards duplicates in list
  uniq_Value(d |> l) = if(d in l, uniq_Value(l), d |> uniq_Value(l));
  uniq_Value([]) = [];

  %collect: applies function ft to every list element
  collect_Value(d |> l, ft) = ft(d) |> collect_Value(l, ft);
  collect_Value([], ft) = [];
```

```
  %collect2: applies function ft2 to every two elements at the same position
  %in the two lists until the end of at least one list is reached
  collect2_Value(d |> l, d' |> l', ft2) = ft2(d, d') |> collect2_Value(l, l', ft2);
  collect2_Value(l, [], ft2) = [];
  collect2_Value([], l, ft2) = [];

  %assert: FAIL_Value is not implemented, so if b does not hold, a rewriting error occurs
  assert_Value(b, d) = if(b, d, FAIL_Value(d));


map
  forall_n: Nat # (Nat->Bool) -> Bool;
  exists_n: Nat # (Nat->Bool) -> Bool;
  get_Values_from_Ciphertexts: List(Ciphertext) -> List(Value);
  get_ValueLists_from_Ciphertexts: List(Ciphertext) -> List(Values);
  FAIL_ASSERT: Bool;
  assert: Bool -> Bool;
var
  n, i: Nat;
  fn: Nat -> Bool;
  a, a', v, v': Value;
  vs, vs': List(Value);
  l, l': List(Value);
  lc: List(Ciphertext);
  c: Ciphertext;
  b: Bool;
eqn
  %forall and exists over indices from 0 to n
  forall_n(0, fn) = true;
  (n>0) -> forall_n(n, fn) = fn(Int2Nat(n-1)) && forall_n(Int2Nat(n-1), fn);
  exists_n(0, fn) = false;
  (n>0) -> exists_n(n, fn) = fn(Int2Nat(n-1)) || exists_n(Int2Nat(n-1), fn);

  %assert: FAIL_ASSERT is not implemented, so if b does not hold, a rewriting error occurs
  assert(b) = if(b, b, FAIL_ASSERT);

  %auxiliary function that turns a list of ciphertexts into a list of values
  get_Values_from_Ciphertexts(c |> lc) = _vs(c) ++ get_Values_from_Ciphertexts(lc);
  get_Values_from_Ciphertexts([]) = [];

  %auxiliary function that turns a list of ciphertexts into a list of lists of values
  get_ValueLists_from_Ciphertexts(c |> lc) = _vs(c) |> get_ValueLists_from_Ciphertexts(lc);
  get_ValueLists_from_Ciphertexts([]) = [];

sort Anno = struct at(_cp: CP, _do: List(CP)) | at_s(_cp_s: CP) | AnnoLeft;
map
  AnnoDY: Anno;
  AnnoNone: Anno;
var
  vs: Values;
  a, a': Anno;
eqn
  AnnoDY   = at_s(CPDY);
  AnnoNone = at_s(UCP);

sort Values = List(Value);

sort Ciphertext = struct Cpair(_vs: List(Value), _a: Anno);

sort SName = List(Name);

sort SCiphertext = struct SCpair(_ns: List(Name), _cs: List(Ciphertext));
```

```
sort Value = struct
    N (_n: Name)?is_N
  | C (_c: Ciphertext)?is_C
  | SN(_sn: SName)?is_SN
  | SC(_sc: SCiphertext)?is_SC
  | invalid
  | any_name
  | any_ciphertext
  | id
  ;

% more intuitive names for ciphertext elements
map
  key: Ciphertext -> Value;
  els: Ciphertext -> List(Value);
  encrypt: Values # Value # Anno -> Value;
  encrypt: Values # Value -> Value;
  encrypt: Values # Anno -> Value;
  encrypt: Values -> Value;
  set_anno: Value # Anno -> Value;
var
  c: Ciphertext;
  k: Value;
  vs: Values;
  ad, a, a': Anno;
eqn
  key(c) = head(_vs(c));
  els(c) = tail(_vs(c));
  encrypt(vs, k, ad) = C(Cpair(k |> vs, ad));
  encrypt(vs, k)     = C(Cpair(k |> vs, AnnoNone));
  encrypt(vs, ad)    = C(Cpair(vs, ad));
  encrypt(vs)        = C(Cpair(vs, AnnoNone));
  set_anno(C(Cpair(vs, a)), a') = C(Cpair(vs, a'));


map
  _ns: Value -> List(Name);
  _cs: Value -> List(Ciphertext);
  like_N: Value -> Bool;
  like_C: Value -> Bool;
  is_any: Value -> Bool;
  SC_: List(Name) # List(Ciphertext) -> Value;
  addToSC: SCiphertext # Name -> SCiphertext;
  addToSC: SCiphertext # Ciphertext -> SCiphertext;
  addToSC: SCiphertext # SName -> SCiphertext;
  addToSC: SCiphertext # SCiphertext -> SCiphertext;
  addToSC: SCiphertext # Value -> SCiphertext;

  simplify: Value -> Value;
  ensure_valid: Value -> Value;
  is_valid: Value -> Bool;
var
  n, n': Name;
  c, c': Ciphertext;
  sn, sn': SName;
  sc, sc': SCiphertext;
  v: Value;
  ns: List(Name);
  cs: List(Ciphertext);
eqn
  _ns(SC(sc)) = _ns(sc);
  _cs(SC(sc)) = _cs(sc);
  like_N(v) = (is_N(v) || is_SN(v));
  like_C(v) = (is_C(v) || is_SC(v));
  is_any(v) = (v in [any_name, any_ciphertext]);
  SC_(ns, cs) = SC(SCpair(ns, cs));
```

```
  addToSC(SCpair(ns, cs), n) = SCpair(if(n in ns, ns, n |> ns), cs);
  addToSC(SCpair(ns, cs), c) = SCpair(ns, if(c in cs, cs, c |> cs));
  addToSC(SCpair(ns, cs), sn) = SCpair(uniq_Name(ns ++ sn), cs);
  addToSC(SCpair(ns, cs), sc) = SCpair(uniq_Name(ns ++ _ns(sc)), uniq_Ciphertext(cs ++
_cs(sc)));
  addToSC(sc', N(n))   = addToSC(sc', n);
  addToSC(sc', SN(sn)) = addToSC(sc', sn);
  addToSC(sc', C(c))   = addToSC(sc', c);
  addToSC(sc', SC(sc)) = addToSC(sc', sc);



  % simplify simply makes some obvious changes to data representation.
  % behaviour of values on either side should be equal under pmatch, but
  % the right hand side is, generally, easier to read.
  simplify(N(n)) = N(n);
  simplify(C(c)) = ensure_valid(C(c));

  simplify(SN([])) = invalid;
  simplify(SN([n])) = N(n);
  simplify(SN(n |> n' |> sn)) = SN(n |> n' |> sn);

  simplify(SC(SCpair([], [])))  = invalid;
  simplify(SC(SCpair([], [c]))) = C(c);
  simplify(SC(SCpair(n |> ns, c |> c' |> cs)))  = SC(SCpair(n |> ns, c |> c' |> cs));



  ensure_valid(v) = if(is_valid(v), v, invalid);

  is_valid(invalid)  = false;
  is_valid(N(n))   = true;
  is_valid(SN(sn)) = (sn != []);
  is_valid(C(c))   = (_vs(c) != []) && forall_Value(_vs(c), lambda v: Value.is_valid(v));
  is_valid(SC(SCpair([], []))) = false;

  % pmatch should ensure that SC's never contain the value invalid - assert raises a
  % rewrite error or returns true.
(ns != []) || (cs != []) ->
  is_valid(SC(SCpair(ns, cs))) = assert(
                       forall_Ciphertext(cs, lambda c: Ciphertext.is_valid(C(c)))
                     );

sort DecryptResult = struct rm_fail(Anno, Anno)?rm_failed | rm_pass;

map
  can_match: Ciphertext  # Ciphertext # Ciphertext -> Bool;
  can_match: SCiphertext # Ciphertext # Ciphertext -> Bool;
  can_match: SCiphertext # Values     # Values -> Bool;

  can_match: Value       # Ciphertext # Ciphertext -> Bool;
  can_match: Value       # Value      # Value -> Bool;

  replace_ids: Ciphertext # Ciphertext -> Ciphertext;
  replace_ids: Values     # Values     -> Values;

  pmatch:   Value       # Value        -> Value;
  pmatch:   SName       # SName        -> SName;
  pmatch_r: Ciphertext  # Ciphertext   -> Ciphertext;
  pmatch_r: List(Value) # List(Value)  -> List(Value);
  pmatch_r: SCiphertext # Ciphertext   -> SCiphertext;
  pmatch_l: SCiphertext # Ciphertext   -> SCiphertext;
  pmatch_r: SCiphertext # SCiphertext  -> SCiphertext;

  pmatch_cs_c_r:  List(Ciphertext) # Ciphertext  -> List(Ciphertext);
  pmatch_cs_c_l:  List(Ciphertext) # Ciphertext  -> List(Ciphertext);
  pmatch_cs_sc_r: List(Ciphertext) # SCiphertext -> List(Ciphertext);
  pmatch_cs_sc_l: List(Ciphertext) # SCiphertext -> List(Ciphertext);
```

```
  can_make_c_r: SCiphertext # Ciphertext -> List(Ciphertext);
  can_make_c_l: SCiphertext # Ciphertext -> List(Ciphertext);

  can_make_v: SCiphertext # Value       -> Value;
  can_make_v: Value       # SCiphertext -> Value;

var
  q, P, P', v, v': Value;
  LQ, LP, LP', LP'', vs, vs', vs'': List(Value);

  n, n': Name;
  c, c', cp, cp', cp'': Ciphertext;
  sn, sn': SName;
  sc, sc': SCiphertext;
  cs, cs': List(Ciphertext);
  ns, ns': List(Name);
  a, a', ao, ad: Anno;
eqn
%%can_match:  Ciphertext # Ciphertext # Ciphertext -> Bool;
  % performs a pattern match while taking annotations into account.
  % note: the annotation in cp is disregarded.
  can_match(c, cp, cp'')  = C(cp'') == match
  whr
    match = pmatch(C(c), C(Cpair(_vs(cp), AnnoLeft))),
  end;

%%can_match: SCiphertext # Ciphertext # Ciphertext -> Bool;
  % performs a pattern match while taking annotations into account.
  % note: the annotation in cp is disregarded.
  % there will be more than one possibility if more than one C in sc
  % could be matched to cp.
  can_match(sc, cp, cp'') = assert(_ns(match)==[]) && (cp'' in _cs(match))
  whr
    % pmatch without simplification, so that we are certain an SC is obtained.
    match = pmatch_r(sc, Cpair(_vs(cp), AnnoLeft)),
  end;

%%can_match:  SCiphertext # Values # Values -> Bool;
  % performs a pattern match while disregarding annotations, i.e.
  % treating ciphertexts simply as a list of values
  % note: the annotation in cp is disregarded.
  % there will be more than one possibility if more than one C in sc
  % could be matched to vs.
  can_match(sc, vs, vs'') =
    assert(_ns(match)==[]) && (vs'' in get_ValueLists_from_Ciphertexts(_cs(match)))
  whr
    % pmatch without simplification, so that we are certain an SC is obtained.
    match = pmatch_r(sc, Cpair(vs, AnnoLeft)),
  end;

%%can_match forwarders
  can_match(C(c),   cp, cp') = can_match(c,  cp, cp');
  can_match(SC(sc), cp, cp') = can_match(sc, cp, cp');
  can_match(N(n),   cp, cp') = false;
  can_match(SN(sn), cp, cp') = false;
  can_match(v, C(cp), C(cp')) = can_match(v,  cp, cp');

%%pmatch: Value # Value -> Value

  pmatch(N(n), N(n'))      = simplify(SN(pmatch([n], [n'])));
  pmatch(SN(sn), N(n))     = simplify(SN(pmatch(sn,  [n] )));
  pmatch(N(n), SN(sn))     = simplify(SN(pmatch(sn,  [n] )));
  pmatch(SN(sn), SN(sn'))  = simplify(SN(pmatch(sn,  sn' )));

  pmatch(C(c), C(c'))      = simplify( C(pmatch_r(c,  c')));
  pmatch(SC(sc), C(c))     = simplify(SC(pmatch_r(sc, c)));
  pmatch(C(c), SC(sc))     = simplify(SC(pmatch_l(sc, c)));
  pmatch(SC(sc), SC(sc'))  = simplify(SC(pmatch_r(sc, sc')));
```

```
  pmatch(any_name, N(n)) = N(n);
  pmatch(any_name, SN(sn)) = SN(sn);
  pmatch(any_name, C(c)) = invalid;
  pmatch(any_name, SC(sc)) = SN(_ns(sc));
  pmatch(any_ciphertext, N(n)) = invalid;
  pmatch(any_ciphertext, SN(sn)) = invalid;
  pmatch(any_ciphertext, C(c)) = C(c);
  pmatch(any_ciphertext, SC(sc)) = SC(sc);
  pmatch(any_ciphertext, any_name) = invalid;

  is_any(v') -> pmatch(v, v') = pmatch(v', v);

(like_N(v) != like_N(v')) && !is_any(v) && !is_any(v') ->
  pmatch(v, v')    = invalid;

  pmatch(invalid, v) = invalid;
  pmatch(v, invalid) = invalid;


%%pmatch: SName # SName -> SName;
  %compute simple intersection
  pmatch(n |> sn, sn') = if(n in sn', n |> pmatch(sn, sn'), pmatch(sn, sn'));
  pmatch([], sn') = [];


%%pmatch: Ciphertext  # Ciphertext  -> Ciphertext;
  pmatch_r(Cpair(vs, ad), Cpair(vs', ao)) =
    Cpair(pmatch_r(vs, vs'), if(ao==AnnoLeft, ad, ao));

%%pmatch: List(Value) # List(Value)  -> List(Value);
  % collect the pmatch of every two elements in the lists.
  pmatch_r(v |> vs, v' |> vs') =
    if(is_valid(pvv) && recurse != [invalid],
      pvv |> recurse,
      [invalid]
    )
    whr
      pvv = pmatch(v, v'),
      recurse = pmatch_r(vs, vs')
    end;

  pmatch_r([], v |> vs) = [invalid];
  pmatch_r(vs, []) = if(vs==[], [], [invalid]);
  % cannot do pmatch([],[]) because of ambiguity


%%pmatch: SCiphertext # Ciphertext  -> SCiphertext;
  pmatch_r(sc, c) = SCpair([], pmatch_cs_c_r(_cs(sc), c) ++ can_make_c_r(sc, c));
  pmatch_l(sc, c) = SCpair([], pmatch_cs_c_l(_cs(sc), c) ++ can_make_c_l(sc, c));


%%pmatch_cs_c: List(Ciphertext) # Ciphertext -> List(Ciphertext)
  % finds all the c's in cs that can be matched with c, and returns the matches in a list
  %
  % _r-version guarantees that the annotation of the second argument is preserved
  pmatch_cs_c_r(c |> cs, c') =
    if(is_valid(C(pcc)),
      pcc |> recurse,
      recurse
    )
    whr
      pcc = pmatch_r(c, c'),
      recurse = pmatch_cs_c_r(cs, c')
    end;
```

```
  % _l-version guarantees that the annotation of the second argument is preserved
  % does not differ from _r-version except the r/l'ness of the subroutines called
  pmatch_cs_c_l(c |> cs, c') =
    if(is_valid(C(pcc)),
      pcc |> recurse,
      recurse
    )
    whr
      pcc = pmatch_r(c', c),
      recurse = pmatch_cs_c_l(cs, c')
    end;

  pmatch_cs_c_r([], c') = [];
  pmatch_cs_c_l([], c') = [];

%%can_make_c: SCiphertext # Ciphertext -> List(Ciphertext)
  % matches each v in c' to either sc's names or sc itself, or returns the empty list if
  % at least one v cannot be matched. returns a list with one or zero elements.
  %
  % _r-version guarantees that the annotation of the second argument is preserved
  can_make_c_r(sc, Cpair(v |> vs, a)) =
    if(is_valid(cmv) && recurse != [],
      [Cpair(cmv |> _vs(recurse.0), if(a==AnnoLeft, AnnoDY, a))],
      []
    )
    whr
      cmv = can_make_v(sc, v),
      recurse = can_make_c_r(sc, Cpair(vs, a))
    end;

  % _l-version guarantees that the annotation of the second argument is preserved
  % differs from _r-version in order of can_make_v arguments, and that the DY annotation
  % is always appended, because this ciphertext has been created from names in the SC;
  % these names must have been known by the attacker, so it is in fact a ciphertext sent
  % by the attacker.
  can_make_c_l(sc, Cpair(v |> vs, a)) =
    if(is_valid(cmv) && recurse != [],
      [Cpair(cmv |> _vs(recurse.0), AnnoDY)],
      []
    )
    whr
      cmv = can_make_v(v, sc),
      recurse = can_make_c_l(sc, Cpair(vs, a))
    end;

  can_make_c_r(sc, Cpair([], a)) = [Cpair([], a)];
  can_make_c_l(sc, Cpair([], a)) = [Cpair([], a)];

%%can_make_v: SCiphertext # Value -> Value;
%%can_make_v: Value # SCiphertext -> Value;
  % properly matches a value against a sciphertext, depending on the value's type.
  % always preserves the annotation of the second argument (unless is_any(v))
  can_make_v(sc, v) =
    if(is_any(v),
      pmatch(v, SC(sc)),
      if(like_C(v),
        pmatch(SC(sc), v),
        pmatch(SN(_ns(sc)), v)
      )
    );

  can_make_v(v, sc) =
    if(is_any(v),
      pmatch(v, SC(sc)),
      if(like_C(v),
        pmatch(v, SC(sc)),
        pmatch(v, SN(_ns(sc)))
      )
    );
```

```
%%pmatch: SCiphertext # SCiphertext -> SCiphertext;
  pmatch_r(SCpair(ns, cs), SCpair(ns', cs')) = %SCpair([],[]);
    SCpair(
      pmatch(ns, ns'), %intersect names
      uniq_Ciphertext( pmatch_cs_sc_r(cs, SCpair(ns', cs')) ++
        pmatch_cs_sc_l(cs', SCpair(ns, cs)) )
    );


%%pmatch_cs_sc: List(Ciphertext) # SCiphertext -> List(Ciphertext)
  % finds all the c's in cs that can be matched with sc, and returns the matches in a list
  pmatch_cs_sc_r(c |> cs, sc) =
    if(is_valid(SC(pc)),
      _cs(pc) ++ recurse,  % ok to discard _ns(pc) because pmatch: SC#C does not fill it.
      recurse
    )
    whr
      pc = pmatch_l(sc, c),
      recurse = pmatch_cs_sc_r(cs, sc)
    end;

  pmatch_cs_sc_l(c |> cs, sc) =
    if(is_valid(SC(pc)),
      _cs(pc) ++ recurse,  % ok to discard _ns(pc) because pmatch: SC#C does not fill it.
      recurse
    )
    whr
      pc = pmatch_r(sc, c),
      recurse = pmatch_cs_sc_r(cs, sc)
    end;

  pmatch_cs_sc_r([], sc) = [];
  pmatch_cs_sc_l([], sc) = [];


sort Message = List(Value);

map
  matchDO: Anno#Anno -> Bool;
  fails_rm : Anno#Anno -> Bool;
var
  o, d: CP;
  O, D: List(CP);
  ad, ao: Anno;
eqn
  matchDO(at(d, D), at(o, O)) = (d in O) && (o in D);
  matchDO(at_s(d),  at(o, O)) = (d in O);
  matchDO(at(d, D), at_s(o))  = (o in D);
  matchDO(at_s(d),  at_s(o))  = true;

  fails_rm(ad, ao) = !matchDO(ad, ao);


sort Knowledge = SCiphertext;
sort KnowledgeUpdate = struct KU(_kn: Knowledge, _dc: List(Ciphertext));

map
  update_knowledge: Knowledge # Message -> KnowledgeUpdate;
  update_knowledge: KnowledgeUpdate # Message -> KnowledgeUpdate;

  propagate: KnowledgeUpdate -> KnowledgeUpdate;

  cs_can_decrypt: List(Ciphertext) # SCiphertext -> List(List(Ciphertext));
```

```
var
  ku: KnowledgeUpdate;
  dc: List(Ciphertext);
  kn, kn': Knowledge;
  v, v', k: Value;
  c, c': Ciphertext;
  n, n': Name;
  sc, sc': SCiphertext;
  sn, sn': SName;
  ns: List(Name);
  cs: List(Ciphertext);
  m: Message;
eqn
  update_knowledge(kn, m) = propagate(update_knowledge(KU(kn, []), m));

  update_knowledge(ku, []) = ku;
  update_knowledge(KU(kn, dc), v |> m) = update_knowledge(KU(addToSC(kn, v), dc), m);

%%propagate: KnowledgeUpdate -> KnowledgeUpdate;
  % fixed point recursion: as long as there are undecrypted ciphertexts in the knowledge
  % that can be decrypted, keep doing so.
  %
  % this is suboptimal, because in fact we only need to try to decrypt newly received /
  % decrypted names and ciphertexts. this is much simpler, however.

  propagate(KU(SCpair(ns, cs), dc)) =
    if(cs_key_known_pair.0 == [],
      KU(SCpair(ns, cs), dc),
      update_knowledge(
        KU(SCpair(ns, cs_key_known_pair.1), cs_key_known_pair.0 ++ dc),
        uniq_Value(get_Values_from_Ciphertexts(cs_key_known_pair.0))
      )
    )
    whr
      cs_key_known_pair = cs_can_decrypt(cs, SCpair(ns, cs))
    end;


%%cs_can_decrypt: List(Ciphertext) # SCiphertext -> List(List(Ciphertext));
  % returns a pair of lists of ciphertexts. the first element contains ciphertexts that
  % can be decrypted. the second element contains ciphertexts that cannot be decrypted.
  %
  % loops through the ciphertexts to see if the key can be pmatched to (found in) sc.
  % if so, the pattern matched result replaces the key, and the whole is prepended to the
  % first list in returned list.
  %
  % otherwise, it is appended to the second list.
  %
  % this effectively means that if a ciphertext has a symbolic key which can only be partly
  % matched, the other values represented by the key are discarded. this is allright, because
  % one matched key already suffices for unlocking the other elements of the ciphertext, and
  % an empty ciphertext with only a key has no value to the attacker, because it cannot find
  % that key until it otherwise receives it, somehow.

  cs_can_decrypt(c |> cs, sc) =
    if(is_valid(pk),
      [_c(encrypt(els(c), pk, _a(c))) |> recurse.0, recurse.1],
      [recurse.0, c |> recurse.1]
    )
    whr
      pk = pmatch(key(c), SC(sc)),
      recurse = cs_can_decrypt(cs, sc)
    end;

  cs_can_decrypt([], sc) = [[],[]];


act
  send, recvA, c, s: Message;
```

```
  recv, sendA, r: Message # SCiphertext;
  FAIL: Value # Value # Anno; %Anno#Anno;
  zero;


proc decrypt(x: Value, p: Value, p': Value, ao: Anno) =
  sum ad:Anno.
    can_match(x, p, set_anno(p', ad)) ->
      matchDO(ad, ao) ->
        tau
      <>
        FAIL(x, set_anno(p, ad), ao).delta;


proc read(p: Values, p': Values) =
  sum sc_dy: SCiphertext.can_match(sc_dy, p, p') ->
    recv(p, sc_dy);


map
  filter_ku_rm: KnowledgeUpdate -> KnowledgeUpdate;
var
  kn: Knowledge;
  cs: List(Ciphertext);
eqn
  filter_ku_rm(KU(kn, cs)) =
    KU(kn, filter_Ciphertext(cs, lambda c: Ciphertext.fails_rm(_a(c), AnnoDY)));

proc DYinit(ns: List(Name)) = DY(SCpair(ns, []));

proc DY(kn: Knowledge) =
  (
    sum m: Message.
      recvA(m).
      (
        %single-point domain sum for not having to compute ku twice.
        sum ku:KnowledgeUpdate.(ku==filter_ku_rm(update_knowledge(kn, m))) ->

          % list was empty? good, the attacker did not decrypt anything it wasn't allowed to.
          (_dc(ku)==[]) ->
            DY(_kn(ku))
          <>
            % otherwise, we raise a FAIL for every not-allowed decryption by the attacker.
            (
              sum c: Ciphertext.(c in _dc(ku)) ->
                FAIL(C(c), any_ciphertext, AnnoDY).delta
            )
      )
  )
+
  (
    sum c_dy: Values.
      sendA(c_dy, kn).
      DY(kn)
  );

map
  CPDYonAttackerIndex: List(Nat) -> List(CP);
  attackerIndex: Nat;
var
  ln: List(Nat);
  n: Nat;
eqn
  CPDYonAttackerIndex([]) = [];
  CPDYonAttackerIndex(n |> ln) = if(n==attackerIndex, [CPDY], CPDYonAttackerIndex(ln));


% end of preamble.
```