Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di Laurea

Security protocol verification by means of symbolic model checking.

Candidato

Giacomo Baldi

Relatori

Prof. Gianluigi Ferrari
Dott. Andrea Bracciali
Dott. Emilio Tuosto

Controrelatore

Prof. Pierpaolo Degano

Anno Accademico 2002-2003

Contents

Chapter 1

Introduction

Since from the birth of the Internet, it has been necessary to achieve *secure* communications among the net end-points. Indeed, users of distributed systems send their private information across a public channel, exposing their data to the risk of being stolen, modified or compromised. Security in distributed systems is not only related to *secrecy* or *integrity* of the data, but also to properties of the interaction between participants to the protocol (also called *principals*). For instance, it is necessary to enable two or more users willing to start a communication, to be able to determine each other identity by means of *authentication* mechanisms. Recently, due to the growth of electronic commercial transactions, more complex desired interaction properties are needed. For instance, to ensure that no parties can get advantages on the others involved in the protocol, *fairness* in digital contract signing is required.

Protocol verification is not only a fascinating theoretical problem but it is also a problem of practical importance: Security protocols allow for secure communications that are an essential feature of many distributed application. Nowadays electronic commerce and commercial transactions are carried over through the network and more and more enterprises are introducing their business into the Internet. Moreover, the new and flourishing field of mobile applications is requesting security protocols to suit their needs. Protocol certification is therefore a critical task that must ensure the safety of the interaction environments on which much of the world economy and inter-communication depends.

Security protocols have been developed to provide secure communications. In general, a protocol is the definition of the format and the order of pieces of information exchanged among principals such that, at the end of the protocol, a desired property holds. Protocols rely on *cryptography* to give principals the capability to transmit encoded information that can be decoded and accessed by the designated receivers only. Cryptography has provided numerous algorithms of data encryption suited for different tasks (e.g. symmetric and asymmetric cryptosystems), which in turn have been used in many different protocols.

Protocols can be *attacked* by hostile entities, usually called *intruders*, which manage to subvert the aims of the protocol, making "honest" principals believe that a certain property holds when it does not. For instance, an attack is performed on a protocol that guarantees authentication, if an intruder can behave under someone else identity without making aware of it other principals. Even well-designed encryption systems do not guarantee the absence of flaws that are not inherent to the cryptographic algorithms, but to the mechanism of the protocol itself. If a protocol is not adequately verified against these situations, then security failures are possible.

The design of cryptographic protocols is hence a complex and difficult task. Indeed, in order to define a robust and correct protocol one has to find a "winning strategy" against an hostile intruder that can alter information flowing through the network at will. Such security flaws can be difficult to discover and generally counter-intuitive. Moreover, protocol verification is a hard problem because, even if protocols are "small" objects consisting of few message exchanges, they give rise to a large number of possible interactions. The attacker can also interfere with exchanged data in many different way. These observations motivate a consistent interest for formal and possibly automated verification methodologies.

1.1 Security protocol analysis

In the analysis of a cryptographic protocol, one is interested in discovering errors which are caused by:

- Flaws or peculiarities of the used cryptosystem,
- bad definition of the interactions between principals.

In this thesis we are interested in the second type of faults that are not necessarily dependent on the underlying cryptographic mechanisms.

Traditionally, cryptographic protocols have been verified using informal and intuitive techniques. This approach is not satisfactory because in such analysis many subtleties are not taken into consideration, preventing security errors to be discovered. For instance, the public-key authentication protocol of Needham and Schroeder [33] was considered secure for over a decade but, after a formal verification, a mis-functioning has been exposed [28]. This highlights the fact that, even for simple protocols, an informal analysis may be too error-prone to be reliable.

Formal analysis of protocols is a complex task and is usually performed in two steps:

- The behaviour of principals is formalised by means of a suitable framework, and
- a formal representation of the security property to check is given.

Modeling protocol semantics is challenging because principals act in a hostile environment. Indeed, a good model must formalise to some extent the intruder's capabilities, which are typically rich enough to make informal verification unfeasible. Moreover, the model must abstract from the details of the data exchanged (which in real protocols are strings of bits) without loosing peculiar features that can be exploited for some kind of attacks. Finally, protocols are meant to run in a distributed and concurrent environment and it is possible that different executions of the same protocol are running at the same time with principals playing different roles in different runs. Such situations are difficult to be modeled correctly, because the number of possible interactions between principals and intruders is very large.

Property formalisation is also hard because it implies the clear understanding of protocol requirements and goals, which in general are not precisely stated in the protocol specification. Moreover, the intuitive idea of the security property usually contains assumptions on the principal behaviour or intruder capabilities that are not made explicit in the specification. Obviously, the lack of such assumptions can make the verification process pointless. Some security properties may be very complex because they involve conditions on the behaviour of a principal with respect to another one and to data exchanged between them, like authentication (i.e. being able to recognise the real identity of the other principals). Adding the fact that the properties predicate about a complex distributed system, their formalisation becomes harder. Indeed, a property specified with respect to the interaction among a certain number of principals, may be non pertinent when more than two principals are involved.

Formal methods can simplify the task of security protocol design and verification in several ways:

- They remove ambiguity in the specification to eliminate common misunderstanding;

- they clearly identify the properties a protocol has to satisfy and the assumptions on the environment under which they hold;
- they can provide insights on the potential weaknesses of protocols;
- they provide tools both for specification and verification.

Section 1.2 will overview some of the most known formal verification techniques and will discuss how they work.

1.2 Formal verification techniques

The general problem of the correctness of a security protocol is undecidable. This result, presented in [16], stems from three sources of infiniteness:

(*i*) Principals and intruders can have the computational power of a Turing machine that can present a non terminating behaviour,

(*ii*) there can be arbitrary many sessions of the same protocol which can be interleaved in an arbitrary way,

(*iii*) the intruder can send to a principal waiting for some data, an infinite number of different messages.

Despite these limitations, there exist many different techniques to achieve correctness results under weaker but reasonable assumptions. We can identify three main approaches: Theorem proving, static analysis, and model checking.

Theorem proving is based on logic and type theories. Generally speaking, the principal behaviour is formalised as a set of clauses representing statements on principal data and a set of inference rules to capture message exchange mechanisms. Properties are also formalised as clauses predicating on the intruder capabilities. The verification process includes a deduction phase that builds a model with a finite number of rules of inference that grow in a controlled manner. Verification is reduced to check if the desired properties belongs to the model, i.e. it can be deduced from the initial assumptions. Theorem proving can discover flaws of protocols with an unbounded number of principal instances and arbitrary complex data. The big drawback is that the procedure is in general semi-decidable. Indeed, termination depend on policies for rule application and protocol characteristics. Hence, if an attack exists, the verification terminates in a finite time, otherwise it might not halt. Therefore theorem provers are often semiautomatic and they require human intervention. Moreover, only the existence of an attack is reported and there is no feedback on how it is performed. However, there are some classes of interesting protocols for which theorem proving procedure always terminate (finite models). Theorem provers have been developed for protocol verification by Paulson [35] and by Blanchet [4].

Static analysis has been traditionally used to verify properties of programs. In general, static analysis techniques are based on an approximation of the system to be verified, on which it is easier and faster to check a property that hence is proven true regardless of the actual data flowing through the system. Recently, in [7, 5], static analysis techniques have been applied to protocol analysis. Principals are represented as *processes* of a suitable *calculus* running in a distributed environment. Security properties are embodied into the principal descriptions by means of *annotations*. Each annotation is placed in protocol control points and restricts the type of data that is allowed to flow through them. Static analysis computes sets of data that can flow through each control point, and property verification is reduced to check whether they contain data that violates annotations. Other similar approaches are based on type inference as in [22]. Static analysis techniques are very efficient and can prove correctness of protocols with unbounded instances. On the contrary, false errors can be found, because computed sets of data are indeed super-sets of the data that actually flows. Moreover, as in theorem proving, no counterexample is given and the specification of annotations is a task that requires a strong expertise.

Model checking is an automatic technique that have been used for the verification of finite state machines. It is also well suited for verifying security protocols, and is based on building a finite model of the environment and checking that a desired property holds in that model. Traditionally models are transition systems and properties are expressed in some modal logic. The property to be verified is checked on each generated state until a state violating the property is found. Due to its finiteness, model checking cannot be used to verify protocols with unbounded number of instances. Moreover some limitations must be imposed on messages to keep finite the number of traces.

Usually, in model checking security protocols, principals are represented as concurrent processes gathered in an hostile environment, that capture the *state* notion, on which transition rules are defined. The transition system representing all the possible executions (also called *traces*) of the protocol is required to be finite. Properties are expressed by means of a suitable logic and are checked for satisfability in every state of the system. The computation starts from an initial state and, by applying the transition rules, it generates the whole state space. Recently in [24], in order to cope with arbitrary complex messages, *symbolic* techniques have been introduced to allow the verification of infinite state systems. The basic idea of symbolic techniques is to avoid the explicit representation of all principal communication, grouping them together when possible. Indeed, it is possible to represent an infinite set of transitions, relative to an infinite set of arbitrary complex messages, by means of a single transition labeled with a characterization of such set. Hence, a symbolic state represents an infinite set of concrete states that can be obtained by instantiating symbolic messages in the symbolic state. Since model checking deals with protocol traces, whenever an attack is found a counterexample can be reported, giving much insight on what has gone wrong. Another advantage of model checking is that it can be fully automated and requires not much domain knowledge to be applied successfully. The main drawback is the *state space explosion* problem. Indeed, since the number of traces can be very large, due to principal action interleaving and number of possible messages, the resulting transition system becomes very big even for a small number of principal instances. Bounding the instances of principals, the problem becomes decidable even if NP-Complete [37]. Indeed, the main challenge in model checking is the development of efficient and smart data structure to handle large state spaces. Recently, some effort has been spent on trying to show under what circumstances checking the protocol with a finite number of instances, may be sufficient to prove correctness [29] in the unbound case.

1.3 $\mathcal{A}$SPASyA: A symbolic model checker

In this thesis we present $\mathcal{A}$SPASyA which is an **A**utomatic tool for **S**ecurity **P**rotocol **A**nalysis based on a **Sy**mbolic model checking **A**pproach. $\mathcal{A}$SPASyA is based on the theoretical work presented in [10, 41], that introduces cIP, a formal calculus for the modeling of cryptographic agent interactions in open systems together with the $\mathcal{PL}$ logic for the specification of security properties and a symbolic semantics for the generation of the symbolic state space. The aim of our tool is twofold:

(i) To develop a usable and flexible tool that supports a methodology for incremental analysis based on successive refinements of protocol assumptions, and

(ii) to implement an efficient model checker by adding user-guided pruning mechanism to the standard algorithm, to reduce the state space size.

Regarding (i), we developed a verification methodology which tries to simplify the precise formalisation of the informal protocol specification. This can be done by an incremental process during which protocol assumptions are refined by the analysis of verification results. Moreover, it is possible to tune the search, and hence the accuracy of the verification, by defining the power of the intruders and the kind of connections between principals. The former is done by specifying the information known by the intruder before the protocol starts (i.e. previously exchanged messages, stolen secrets, etc.), whereas the latter is realised by constraining the kind of possible connections between principals, hence pruning the state space. It is important to note that the ease of use and the high degree of automation are features provided by $\mathcal{A}$SPASyA. Indeed, the application of the methodology requires the specification of the protocol and the security property, whereas the verification process is fully automatic, greatly reducing human interaction. To remark this point we will show in the following how $\mathcal{A}$SPASyA can report some attacks recently discovered by means of static analysis techniques (as presented in [6]) in the Beller-Yacobi protocol [3].

cIP principals can *join* a session of a protocol and *connect* themselves to other instances as specified in [10, 41]. We extended this approach by adding a mechanism to constraint the possible ways principals can connect with each other. As it will be shown , this has a great impact in terms of efficiency because specifying constraints on the joining of principals greatly limits the number of states generated by the operational semantics. This mechanism not only enhances the performance of $\mathcal{A}$SPASyA but also enables the user to perform search on portions of the entire state space, allowing for an easier analysis of results. Another important aspect of $\mathcal{A}$SPASyA is its modular architecture. Indeed, it has been developed for the Profundis[1] project, having in mind interoperability with other tools and, most important, future expansions to add new functionalities.

We performed numerous tests on $\mathcal{A}$SPASyA by applying the verification methodology to some well-known protocols, which will be illustrated together with performance measurement in the following.

1.4 Chapters description

This thesis is organised as follows:

- In chapter 2, we present in more details security protocols. We introduce the commonly used informal specification for the the description of protocols and interesting security properties. We finally report a widely accepted modelisation of the intruder capabilities in the hostile environment of the network presented in [19].
- In chapter 3, we sketch the theoretical work in [10, 41], introducing the cIP calculus and the logic for property specification.
- In chapter 4, we start the presentation of $\mathcal{A}$SPASyA by giving an overview of its modular architecture and of its basic modules.
- In chapter 5, we continue the description of the more complex modules and functionalities of $\mathcal{A}$SPASyA and we introduce a symbolic verification algorithm. We also describe the constrained join mechanism and its effects on the verification procedure.
- In chapter 6, we outline the methodology by applying it to some well-known protocols, illustrating the benefits of our approach.
- In chapter 7, we report a library of protocols with the corresponding translation from their informal specification in our framework, and the relative verification results.
- In chapter 8, we make a comparison between $\mathcal{A}$SPASyA and other similar tools.
- In chapter 9, we report the user manual containing the description of $\mathcal{A}$SPASyA usage.

[1] Refer to `www.it.uu.se/profundis/` for more information.

- In chapter 10, we sum up our results and present future lines of development.
- In the appendix we report the source code of $\mathcal{A}$SPASyA with comments for the most important modules.

Chapter 2

Security Protocols

This section briefly reviews some elementary notions on cryptography, protocol specification, security properties and the attacker model. Its main purpose is to introduce notation and terminology. We refer to [38, 30] for a comprehensive introduction to this field.

2.1 Cryptography

Security protocols have been used to protect "sensible" information in a scenario where two (or more) partners (sometimes called *principals*) communicate by exchanging messages through "public channels". Usually, the word "sensible" means information that should be kept secret or non-modifiable (e.g. a credit card number, an encryption key, etc.). The aim of a protocol is to grant that no malicious participants (usually called *intruders* or *attackers*) will ever be able to disclose or modify sensible information. Certification (specification and verification) of security protocols requires a careful definition of:

- the underlying assumptions adopted in the algorithms used to encrypt/decrypt messages,
- the hypothesis on the capabilities of malicious participants to interfere with the communications.

Cryptography is used to hide the sensible information contained in messages flowing through the public channel. An *intelligible* message m, is referred to as *plaintext* (or *datagram*). By 'intelligible' we mean that the representation of the information denoted by m is public domain knowledge. Viceversa, an *unintelligible* form of m is said *ciphertext* (or *cryptogram*).

The process of assigning a ciphertext to a plaintext is called *encryption*; encryption is parameterised with respect to an *encryption key*. Given a ciphertext, the operation that reconstructs the plaintext form is called *decryption*; as for encryption, decryption has a *decryption key* as parameter. Hereafter, $\{m\}_k$ denotes the cryptogram obtained by encrypting message m with key k, while m, n denotes the pair made of messages m and n. Given k, *decryption* extracts m from $\{m\}_k$.

Crypto-systems can be *symmetric* or *asymmetric*. The former (also known as *private key crypto-systems*) are characterized by the fact that encryption and decryption keys are the same. Indeed two principals, say A and B, can encrypt/decrypt data if they share a key k. It is usually assumed that k is known only by A and B and other principals may acquire k only if A or B explicitly send it. Asymmetric (a.k.a. public key) cryptography is characterized by the fact that encryption and decryption keys differ each other. Each principal A has a *private* key and a *public* key, respectively denoted by A^- and A^+. For each principal A, its public key A^+ is publicly available and may be used by any other principal to encrypt messages intended for A. Such cryptograms may be decrypted only using the private key A^- that only A owns.

We will refer to *symmetric* and *asymmetric* cryptosystems. The robustness of a system relies on the security of various levels of its architecture and their relationships. For the sake of protocol certification, at cryptographic level, the standard working assumption is the so-called *perfect encryption hypothesis* stating that a cryptogram can be decrypted only using its decryption key and that secrets cannot be guessed, no matter how much information is possessed [1].

2.2 Protocol specification

A *security protocol* may be intuitively thought of as a finite sequence of messages between two or more participants. There is a great variety of specifications mechanisms of protocols and their properties. Some protocols are informally specified mixing natural language and ad hoc notation (for instance, SSL [21], SSH [44], IKE [23] are specified in this style). Protocols are usually presented as a list of message exchanges (also called *narration* description) written as:

$$(n) \quad A \to B: \quad m.$$

The intended meaning of this notation is: "at the n-th step A sends message m to B" where A and B represent two principals of the protocol. Typically A is the *initiator*, B is the *responder*, S is a third-part (usually trusted) server and I denotes the intruder.

Example 2.1 *In this style we can give the informal specification of the Wide Mouthed Frog (WMF) protocol [11]. The intent of WMF protocol is to let A send a fresh [2] session key k^{ab} to B through a trusted server S. Both A and B share two private keys with S (k^{as} and k^{bs}, respectively). The WMF protocol is:*

$$\begin{array}{llcll} (1) & A & \to & S: & A, \{T^a, B, k^{ab}\}_{k^{as}} \\ (2) & S & \to & B: & \{T^s, A, k^{ab}\}_{k^{bs}}. \end{array}$$

First, A encrypts for S the identity of B, the session key k^{ab} and a fresh *time-stamp T^a intended to be used only for a session of the protocol; such names are called* nonces. *By the perfect encryption hypothesis, T^a and k^{ab} cannot be "guessed" by any other participant of the protocol. Then, S forwards k^{ab} and the identity of the initiator to B; freshness of k^{ab} is witnesses by T^s, a nonce generated by S.* ◇

A sequence of message exchanges is not a complete specification for security protocols. For instance, the narration in Example 2.1 do not specifies whether or not only A, B and S are the only principals that know k^{ab}. Informal specification does not represent with enough details common situations: For instance, referring to [11], a protocol may have multiple simultaneous runs and a principal may play different roles in different runs. Moreover data sent across the network are strings of bits and it may be possible that a principal does not recognize the shape of the received message or implicitly assumes that the received data have a given form. Indeed, this is a source for protocol *type flaws*. To cope with this problem it is usually assumed that "messages are typed", namely, they contains enough information for a principal to recognise their shape.

[1] Such hypothesis is not completely realistic; indeed, under it, *cryptoanalysis attacks* cannot be captured. Cryptoanalytic attacks are performed by collecting a great number of cryptograms and then analyzing them for deducing cryptographic keys. However, realistic keys cannot be deduced in polynomial time by intruders that have a given computational capacity.

[2] A key is fresh when is generated by a principal and exchanged in the same execution of the protocol. Moreover it is assumed that the key has never been used in any previous run of the protocol. Key k^{ab} is called a *session* key because it will be used to encrypt messages exchanged between A and B after the execution of the protocol.

2.3 Security properties

Many security properties can be stated for a given protocol. We mainly focus on *integrity*, *secrecy* and *authentication*. Intuitively, a protocol guarantees integrity if, once a datum has been provided, it cannot be altered by any intruder. A protocol guarantees secrecy over a set of data if it is not possible that an intruder will get such data. A protocol guarantees authentication of a user A to a user B if, after running the protocol, B may *safely* assume that A was involved in the protocol run. Secrecy and authentication are closely related. Indeed, before sending secrets to B, A should be sure that he is effectively "speaking" to B. Viceversa, authentication is normally achieved through exchange of some data that the protocol ensure to be created by the intended partner and nobody else.

If secret information must be communicated in an untrusted environment, the protocol must ensure at least that possible eavesdroppers cannot understand them (secrecy), that the partner in the communication is really the intended one (authentication) and that the messages are really forged by the intended participant (message integrity). Secrecy, authentication and integrity are the "elementary" properties that protocols aim at guaranteeing.

Security properties are not uniformly stated and defined. The kind of a property and its adequacy depend on applications. For instance, in an electronic commerce application properties like *fairness* or *non-repudiation* are requested together with secrecy, integrity and authentication. A mandatory feature of a voting system is, for example, *anonymity*.

Example 2.2 *Let us consider again the WMF protocol specification (Example 2.1). A possible requirement of the protocol is the secrecy of k^{ab}, namely, in every session, the value of k^{ab} must be known only by the principals playing the roles of A and B and S. Another requirement is that the protocol guarantees the authenticity k^{ab}, i.e., in every session where B receives the message from S, he must be ensured that,* in the same session*, (i) A has created k^{ab} and (ii) that A asked S to forward the key to B.* ◇

2.4 Intruder model

A formal framework for protocol analysis must declare which assumptions are made on the intruder. The Dolev-Yao model [20] is a widely accepted model. It describes an active intruder as a "principal" that can

- receive and store any transmitted message;
- hide a message;
- decompose messages into parts;
- forge messages using known data.

The only limitation for intruders imposed by the Dolev-Yao model is the assumption of the perfect encryption hypothesis. Recently in [36] it has been shown that the Dolev-Yao intruder, enhanced with the capability of guessing a decryption key with a negligible probability, is as powerful as the original one. The model also assumes that intruders can have some private data, namely information which has not been generated by regular principals, and can "remember" data exchanged in previous runs of the protocols. In particular, an intruder can record all exchanged messages and use them later to attack the protocol.

Since an intruder *á la* Dolev-Yao can intercept any communication, it can be formalised as the execution environment which behaves as the "adversary" of "honest" principals. The environment collects all the sent messages and manipulate them when a principal is waiting for some data.

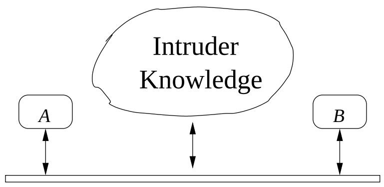

Figure 2.1: A graphical representation of the Dolev-Yao intruder

Let $N = N_o \cup N_p$ be a countable set of names, where N_o is the set of nonces and N_p is the set of principal names; we assume that $N_o \cap N_p = \emptyset$. Let K be a set of keys that contains both symmetric and asymmetric keys and such that $K \cap N = \emptyset$. As usual, we assume that keys are not complex terms, like tuples, but just atomic terms.

A *message* is a term defined as follows:

$$M \quad ::= \quad N \quad | \quad K \quad | \quad M, M \quad | \quad \{M\}_K.$$

A message may be a name (i.e. a nonce or a principal name), a key (symmetric or not), the pairing of two messages or the encryption of a message We let m, n, ... to range over M, while λ ranges over K, λ^- denotes the inverse key of λ, namely, $\lambda^- = \lambda$ if λ is a symmetric key, while $\lambda^- = A^-$ if $\lambda = A^+$ and $\lambda^- = A^+$ if $\lambda = A^-$. A message may be a name (i.e. a nonce or a principal name), a key (symmetric or not), the pairing of two messages or the encryption of a message.

The Dolev-Yao intruder is characterised by a set of messages κ, called the *intruder knowledge*, and the actions that the intruder can perform on the elements of κ. Basically, an intruder can pair known messages, split pairs and encrypt/decrypt cryptograms if the corresponding keys can be deduced by κ. We write $\kappa \triangleright m$ to denote that datum m can be deduced by means of a finite sequence of such actions from messages in κ. The operator $\triangleright$ defines the (infinite) set of messages that an intruder can use for attacking a protocol.

Example 2.3 *Consider the knowledge $\kappa = \{\{A^-\}_k,\ \{m\}_{A^+},\ k\}$, we show how $k \triangleright \{m\}_k$ with the following deduction tree:*

$$\cfrac{\cfrac{\cfrac{\kappa \triangleright \{A^-\}_k \qquad \kappa \triangleright k}{\kappa \triangleright A^-}\,(1) \qquad \kappa \triangleright \{m\}_{A^+}}{\kappa \triangleright m}\,(2) \qquad \kappa \triangleright k}{\kappa \triangleright \{m\}_k}\,(3)$$

where (1) *and* (2) *are decryption by means of a known key ($\kappa \triangleright k$, namely $k \in \kappa$), and* (3) *is encryption.* ◇

In [15] decidability of $\triangleright$ has been proved for private key cryptography; decidability of $\triangleright$ for public key cryptography has been proved in [10, 41].

Chapter 3

Formal Framework

In order to make this thesis as self-contained as possible, this chapter borrows from [10, 41] the approach to certification of security protocols that is based on symbolic state space exploration and model checking. We briefly sketch the formal framework and refer the reader to [10, 41] for further details.

3.1 The cIP calculus

This section presents the main features of the *cryptographic interaction pattern* (cIP) calculus and the *protocol logic* ($\mathcal{PL}$) introduced in [10, 41] to specify and prove properties of security protocols. Apart from constituting the theoretical background of $\mathcal{A}$SPASyA, cIP calculus and $\mathcal{PL}$ logic fix some design choices of the symbolic model checking that are necessary for describing our tool.

The cIP calculus is a name-passing process calculus in the style of the π-calculus [32] with cryptographic primitives. A cIP process is written as $A \triangleq (\tilde{X})[E]$ where A is the process identity, $\tilde{X}$ are the *open variables* and E is a sequence of actions. To illustrate the main features of cIP we consider the WMF protocol.

Example 3.1 *Principals of the WMF protocol are described by the following cIP terms:*

$$\begin{array}{lcl} A & \triangleq & (x, s)[out(A, \{T^a, x, kab\}_s)], \\ S & \triangleq & (u, ak, v, bk)[in(u, \{?t, v, ?r\}_{ak}).out(\{T^s, u, r\}_{bk})], \\ B & \triangleq & (z)[in(\{?y, ?q, ?w\}_z)]. \end{array}$$

$\diamond$

Each process has an identity (e.g. A, S, B) and a sequence of input/output actions; a list of open variables preceeds the actions. Open variables are a distinguished feature of cIP that (together with the *join* operation) provides an explicit mechanisms for sharing names/keys. Open variables bind the free occurrences of variables in the communication actions, like x in A. The occurrences of the variables y, q and w in B are *bound*. Binding occurrences are marked by '?', like $?r$ in S, which binds the following occurrence of r. A datum d of an action is a message (as defined in section 2.4) where variables can appear.

In Example 3.1 the open variable x parameterises the identity of the responder, while s and z are the variables which should be instatiated with keys shared among A and S and B and S respectively. The server S has four open variables: u and ak are reserved for the identity and symmetric key of the initiator, whereas v and bk are used for the identity and the symmetric key of the responder.

An *instance of principal* $A \triangleq (\tilde{X})[E]$ is a process obtained by indexing all the variables (open or not) and all the names in E with a natural number $i > 0$. Instances run in *contexts*, i.e. (possibly empty) sets of instances where computation takes place and new instances may be dynamically added. Many instances of the same principal may (non-deterministically) join the context modeling the execution of more sessions of the protocol. The join operation defines how a principal instance can enter a (running) context by connecting open variables for asymmetric keys to principal names and open variables for symmetric keys to keys K so that they are appropriately shared. Connected variables are no longer open.

We say that a context $\mathcal{C}$ has cardinality n if it contains n instances of principals. We use $\mathrm{ov}(\mathcal{C})$ to indicate the set of open variable of context $\mathcal{C}$.

Definition 3.1 (Join) *Let* $A_n \triangleq (\tilde{X}_n)[E_n]$ *be an instance,* $\mathcal{C}$ *be a context of cardinality* $n-1$ *and* γ *be a partial mapping whose domain is* $\mathrm{ov}(\mathcal{C}) \cup \tilde{X}_n$*. Moreover, for each* $x \in \mathrm{ov}(\mathcal{C}) \cup \tilde{X}_n$*,* $\gamma(x)$ *is either in* K *or in* N_p *depending whether* x *is a variable for a symmetric key or for a principal. The* join *operation is defined as:*

$$join(A_n, \gamma, \mathcal{C}) = (\tilde{X}_n - dom(\gamma))[E_n\gamma] \ \cup \bigcup_{(\tilde{Y})[E'] \in \mathcal{C}} (\tilde{Y} - dom(\gamma))[E'\gamma].$$

Example 3.2 *Consider the cIP template for the WMF protocol (Example 3.1). When* B *joins the context containing* A *and* S*, a possible result of the join operation is given by the following context (for simplicity we ignore indexes):*

$$\begin{array}{lcl} A & \triangleq & (x)[out(A, \{T^a, kas, kab\}_{kas})], \\ S & \triangleq & ()[in(A, \{?t, B, ?r\}_{kas}).out(\{T^s, A, r\}_{k^{bs}})], \\ B & \triangleq & ()[in(\{?y, ?q, ?w\}_{kbs})]. \end{array}$$

It is obtained using $\gamma = \begin{cases} z, bk & \mapsto & kbs \\ s, ak & \mapsto & kas \\ v & \mapsto & B, \\ u & \mapsto & A \end{cases}$ $\diamond$

The concrete semantics of cIP is defined as a reduction relation on *configurations*, reported in table 3.1, which formalizes the behaviour of the Dolev-Yao intruder. Indeed, configurations are triples $\langle \mathcal{C}, \chi, \kappa \rangle$ where:

- $\mathcal{C}$ is a context,
- χ is a set of variable bindings that keeps track of the assignments of the variables due to communications and join executions
- and κ, the intruder knowledge, contains the names of instances that joined the context, the data sent along the public channel, and the optional initial knowledge of the intruder.

Relation $\longmapsto$ models both communications and the possible evolutions of a context due to the joining of new instances. Communications embody encryption and decryption mechanisms and take place by means of pattern matching, indicated as $\sim$.

Example 3.3 *The 'in' action of principal* B *in Example 3.2 waits for a triple encrypted with* kbs*, the symmetric key which* B *shares with* S*. When such a message arrives the elements of the triple are assigned to the corresponding variables. Message* $\{T^s, A, kab\}_{kbs}$ *is a possible matching message for* $\{?y, ?q, ?w\}_{kbs}$*, since the sent message is encrypted with the complementary key (namely the same) of the one used to encrypt the input message. The pattern matching give rise to the substitution* $\gamma = \{y \mapsto T^s, q \mapsto A, w \mapsto kab\}$*.* $\diamond$

$$\frac{\kappa \triangleright m:\ \exists\gamma \text{ ground s.t. } d\gamma \sim m}{\langle(\tilde{X}_i)[in(d).E_i] \cup \mathcal{C}, \chi, \kappa\rangle \ \longmapsto\ \langle(\tilde{X}_i)[E_i\gamma] \cup \mathcal{C}, \chi\gamma, \kappa\rangle}\ (in)$$

$$\frac{\text{—}}{\langle(\tilde{X}_i)[out(m).E_i] \cup \mathcal{C}, \chi, \kappa\rangle \ \longmapsto\ \langle(\tilde{X}_i)[E'_i] \cup \mathcal{C}, \chi, \kappa \cup m\rangle}\ (out)$$

$$\frac{\mathcal{C}' = join(A_i, \gamma, \mathcal{C}) \qquad A \triangleq (\tilde{X})[E] \qquad i \text{ new}}{\langle\mathcal{C}, \chi, \kappa\rangle \ \longmapsto\ \langle\mathcal{C}', \chi\gamma, \kappa \cup \{A_i, A_i^+\}\rangle}\ (join)$$

Table 3.1: Context reduction semantics

$$\frac{\kappa \triangleright m:\ \exists\gamma \text{ symbolic s.t. } d\gamma \sim_{sym} m}{\langle(\tilde{X}_i)[in(d).E_i] \cup \mathcal{C}, \chi, \kappa\rangle \ \longmapsto_{sym}\ \langle(\tilde{X}_i)[E_i\gamma] \cup \mathcal{C}\gamma, \chi\gamma, \kappa\gamma\rangle}\ (in_{sym})$$

Table 3.2: Modified input rule

As we can see from Example 3.3, the matching relation ($\sim$) in the case of symmetric keys simply reduces to require that the encryption keys are equal. For instance, $\{B, kab, T^a\}_{kas}$ matches itself. In the case of asymmetric keys there is a match when cryptograms are encrypted with complementary keys[1] (e.g. $\{A\}_{B^-} \sim \{A\}_{B^+}$). A datum d *matches* a message m if, and only if, there exists a substitution γ over the variables occurring in d such that $d\gamma$ is ground and $d\gamma \sim m$.

Notice that the premise of rule (in) requires the existence of a message m which is derivable from κ, and of a ground substitution γ, such that m matches the datum d that a principal is waiting to receive. The set of such messages (and hence substitutions) may be infinite, leading to a non effective and incomplete verification procedure. Symbolic techniques, introduced recently in [24], provide a powerful mechanism to tackle the problem of infinite branching. Whenever an infinite set of messages can be generated for a binding variable x, we use the *symbolic variable* $x(\kappa)$. This amounts to say that x can assume any message m such that $\kappa \triangleright m$. The value of $x(\kappa)$ will be possibly set by matching later actions.

The semantic rules are then redefined to include symbolic variables; we report the modified (in) rule in Table 3.2. In the symbolic (in_{sym}) rule, we require γ to be a symbolic substitution (i.e. a substitution that contains assignments involving symbolic variables) such that $d\gamma$ *symbolically* matches m ($\sim_{sym}$). The number of such substitutions is finite; moreover, symbolic substitutions are applied to the whole configuration components. Indeed, in the symbolic semantics, contexts and knowledges may also contain symbolic variables that must be instantiated with values determined during previous applications of the symbolic transitions rules. In [10, 41] terminating procedures are introduced to calculate symbolic substitutions required by the premises of semantics rules, in order to make the verification process effective.

The symbolic reduction semantics give rise to symbolic traces, where a trace t is a sequence of configurations $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ such that $\Gamma_i \longmapsto_{sym} \Gamma_{i+1}$ for $0 < i < n$. When principals in the final configuration Γ_n have fired all their actions, we say that Γ_n is a *terminal* configuration and t is a *terminal* trace. Security properties are checked on terminal configurations only; in [10, 41] terminal configurations are models for attacks: This approach characterize those attacks where he intruder is able to deceive principals without being

[1] In cIP asymmetric keys are represented as principal names decorated with + for public keys and − for private keys.

detected. In [10, 41] it has been shown that the symbolic verification procedure is sound, namely if an attack is symbolically found, than at least one correspondent concrete trace exists. Moreover, the verification procedure is also complete (up to the number of principal instances involved) in the sense that the for every concrete trace t there exist a symbolic trace t^s that "contains" t (i.e. there exists a substitution γ such that $t^s\gamma = t$).

3.2 $\mathcal{PL}$ logic

Security properties are expressed as formulae of the $\mathcal{PL}$ logic (Protocol Logic) that allows one to relate values and variables, and assertions on membership of messages to the intruder's knowledge κ.

Definition 3.2 **($\mathcal{PL}$ – Syntax)** *A formula of the logic $\mathcal{PL}$ is defined as follows:*

$$\begin{array}{rcl} \phi, \psi & ::= & \delta \in \mathfrak{K} \;\mid\; \delta = \sigma \;\mid\; \forall A.i : \phi \;\mid\; \neg\phi \;\mid\; \phi \wedge \psi \\ \delta, \sigma & ::= & d \;\mid\; x_i \;\mid\; \mathbf{I} \end{array}$$

where d is a datum that does not contain any binding occurrence, and $\boldsymbol{I}$ is a distinguished constant representing the intruder name.

Operators $\neg$ and $\wedge$ are the usual boolean operators[2]. The symbol $\mathfrak{K}$ is used to represent the knowledge of the intruder. Formula $\delta \in \mathfrak{K}$ states the derivability of δ from $\mathfrak{K}$, whereas $\delta = \sigma$ tests for equality of data. Definition 3.2 introduces quantification over instances of roles. Variables are therefore indexed (e.g. x_i) and indexes are "typed" by principal names. For instance, the proposition $\forall A.i : \phi$ is read as "for all instances of A, ϕ holds", and i may occur in ϕ. Among the possible values that can be expressed in $\mathcal{PL}$ formulae there is the distinguished constant $\mathbf{I}$ that denotes the intruder's identity. This permits us to express propositions where the identity of the principals is not necessarily a "regular" role.

In $\mathcal{PL}$ integrity can be expressed by fixing some values (e.g. $x_i = \delta$), secrecy is handled by values that κ may derive (e.g. $d \in \mathfrak{K}$), and authentication is expressed as relations among principals' variables and communicated messages.

Example 3.4 *A property that the WMF protocol should satisfy is the secrecy of the session key k^{ab}, unless it is really intended for* $\mathbf{I}$*:*

$$\forall A.i : x_i \neq \mathbf{I} \rightarrow kab_i \notin \mathfrak{K},$$

that captures the intuitive secrecy property above. $\diamond$

Formulae are verified with respect to a given (terminating) context of a computation which keeps trace of all the principal instances that participated in the session. Notation $\kappa \models_\chi \phi$ indicates that κ, under the variable assignment χ, is a model of the formula ϕ; κ and ϕ are relative to a final configuration of a terminal trace. We define models for *closed* $\mathcal{PL}$-formulae. Let χ be a ground substitution for variables X. A *model for a closed formula* ϕ is a pair $\langle \kappa, \chi \rangle$ such that $\kappa \models_\chi \phi$ can be proved by the following rules:

$$\frac{i = j}{\kappa \models_\chi A_i = A_j}\,(=_1) \qquad \frac{x_i\chi = \delta\chi}{\kappa \models_\chi x_i = \delta}\,(=_2) \qquad \frac{\kappa \triangleright \delta\chi}{\kappa \models_\chi \delta \in \mathfrak{K}}\,(\in) \qquad \frac{\kappa \not\models_\chi \phi}{\kappa \models_\chi \neg\phi}\,(\neg)$$

$$\frac{\kappa \models_\chi \phi \quad \kappa \models_\chi \psi}{\kappa \models_\chi \phi \wedge \psi}\,(\wedge) \qquad \frac{\kappa \models_\chi \phi[{}^{A_j}/_{\alpha}] \quad \text{for all } A_j : \kappa \triangleright A_j}{\kappa \models_\chi \forall A.\alpha : \phi}\,(\forall).$$

[2]Derived relations $\neq$ and $\notin$, logical connectors $\rightarrow$ and $\vee$, or existential quantifier $\exists$ are defined as usual and will be used as syntactic sugar.

Rule ($=_1$) says that $\langle\kappa,\chi\rangle$ is a model of equality $A_i = A_j$ whether the instances are exactly the same instance. Rule ($=_2$) says that $\langle\kappa,\chi\rangle$ is a model of $A_i = \delta$ whether the value associate by χ to variable x of instance A_i, i.e. $x_i\chi$ equals the value $\delta\chi$. Rule ($\in$) establishes that $\kappa \models_\chi \delta \in \mathfrak{K}$ whenever $\delta\chi$ can be constructed from the decomposition set of κ. Rules ($\neg$) and ($\wedge$) are straightforward. In $\forall A\alpha : .\phi$ the universal quantifier ranges over the finite set of instances of role A. Quantifiers are solved by mapping variables indexes to actual instances. In order to prove that $\langle\kappa,\chi\rangle$ is a model for a formula $\forall A.\alpha : \phi$, it is necessary to show that $\langle\kappa,\chi\rangle$ is a model for any formula obtained by substituting A_j for α in ϕ, where A_j is any instance of A deducible from κ, that participate in the protocol execution.

Notice that if χ and χ' differ only on variables not appearing in ϕ, then $\kappa \models_\chi \phi \Leftrightarrow \kappa \models_{\chi'} \phi$. Hence, we can only consider finite assignments over the variables of ϕ.

Example 3.5 *Let us consider the following authentication property, relative to the WMF protocol*

$$\begin{aligned}\psi = \quad & \forall B.j : \exists S.l : \exists A.i : \\ & (v_l = B_j \wedge\ u_l = A_i \wedge\ x_i = B_j) \rightarrow \\ & (t_l = T_i^a \wedge\ y_j = T_l^s \wedge\ w_j = kab_i \wedge q_j = A_i)\end{aligned}$$

The formula states that, whenever B terminates, a server S and an initiator A (that aimed at interacting with B through S , i.e. $x_i = B_j$) have also took part to the session. In this case, the nonce received by S is the one generated by A ($t_l = T_i^a$), while B receives the nonce generated by S ($y_j = T_l^s$). Finally, the session key received by S must be the key associated to T^a by A ($w_j = kab_i$).

Assume $\kappa_1 = \{A, B, S, I, \{T^a, B, kab\}_k, \{T^s, A, kab\}_k\}$, as the intruder knowledge and, referring to Example 3.2,

$$\gamma = \begin{cases} z, bk & \mapsto & kbs \\ s, ak & \mapsto & kas \\ v & \mapsto & B, \\ u, q & \mapsto & A, \\ r, w \mapsto & kab & \\ t, y \mapsto & T^a & \end{cases}$$

as the set of variable bindings (we ignore indexes for clarity). We show that the final configuration $\langle\{()[], ()[], ()[]\}, \kappa_1, \gamma\rangle$ does not yield a model of ψ. We rewrite ψ according to the ($\forall$) rule expanding quantifiers and obtaining

$$\psi' = (v = B \wedge u = A \wedge x = B) \rightarrow (t = T^a \wedge y = T^s \wedge w = kab \wedge q = A)$$

Under mapping γ, we can see that the antecedent of ψ' holds while the consequent is false due to the equality $y = T^s$. Indeed, $\gamma(y) = T^a \neq T^s$, hence $\kappa_1 \not\models_{\gamma_1} \psi$. $\diamond$

Chapter 4

$\mathcal{A}$SPASyA: Architecture and Data Structures

This chapter shows how the symbolic model has been exploited as a basis for the design and development of an effective and usable verification toolkit called $\mathcal{A}$SPASyA. We will focus on architectural issues and design aspects to highlight the problems we encountered while implementing the symbolic model. $\mathcal{A}$SPASyA has a modular architecture, where each module encapsulates a single aspect of the symbolic model. We will describe the functionality of all the modules and their most important implementation details.

4.1 Architecture

The architecture of $\mathcal{A}$SPASyA is displayed in Figure 4.1 and it is made of three main parts. Incoming arrows indicate input data whereas outcoming arrows indicate outputs. The module `configuration-hdl` manages cIP configurations (and, therefore, $\mathcal{PL}$ models)

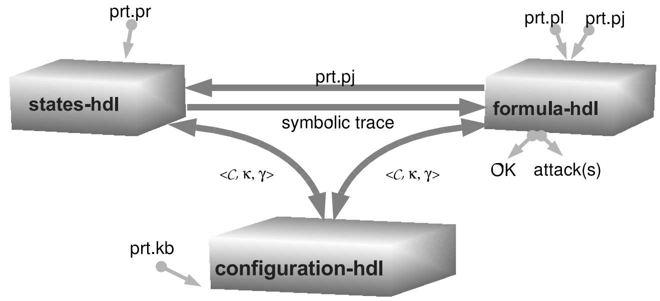

Figure 4.1: $\mathcal{A}$SPASyA architecture

and consists of three components:

- `context` that represents the running principal instances,
- `knowledge` that manages the set of messages in the configurations, and
- `assignments` that handles variable substitutions.

The module `states-hdl` implements the state space generation and contains components:

- `step` that returns the next configuration given the current one, and

- `join` that handles the introduction of new principals in the configuration.

Finally, given a terminal configuration provided by `states-hdl`, `formula-hdl` implements the check of $\mathcal{PL}$ formulae. In order to check formulae it might be necessary to solve systems of symbolic constraints, as clarified in Section 5.5. The components of `formula-hdl` are:

- `logic` that transforms a $\mathcal{PL}$ formula into an equivalent normalised formula,
- `verifier` that checks whether the given configuration is a model for the formula,
- `csolver` that collects constraints on symbolic variables and resolves equality/inequality of symbolic messages.

In a verification session, the user provides the cIP specification of the protocol (file `prt.pr`) equipped with the connection formula (file `prt.pj`) and the $\mathcal{PL}$ formula expressing the security property (file `prt.pl`). Connection formulae will be described in details in Section 5.3.1 and 6.1, but intuitively they are mechanisms to constraint the way principals can join a configuration, by specifying relations among open variables. An initial knowledge (file `prt.kb`) and the maximum number of instances that can join a context can be specified, as well.

Modules `step` and `join` ask to `configuration-hdl` for the current configuration and, according to cIP symbolic semantics and to connection formula, a new configuration is produced and returned back to `configuration-hdl`. Iterating this process, `states-hdl` eventually receives a terminal configuration and forwards the corresponding $\mathcal{PL}$ (symbolic) model to `verifier` that together with `csolver` checks for the validity of (the normalisation of) `prt.pl`. Modules `verifier` and `csolver` return `OK` when the formula holds, otherwise they yield the (possible) attack(s).

4.2 Choosing the language

The first choice to face when developing a piece of software is which language to use for the implementation. In general, there are some aspects related to programming languages to focus on:

- The expressive power, which is related to the level of the language.
- The efficiency of the object code.

While a low-level language might be suitable to write hardware drivers, it may be less suitable for high level programming. A language abstracting away from machine-dependent issues is the language of choice. Moreover model checking is a heavy computational task, so having efficient and fast compiled code is an important feature of the language.

Usually debugging of a model checker is hard because even small tests generate a space state too big to be checked by hand; so the presence of powerful debuggers, precise profilers, code browsers and parser generators is of some moment.

Following this observations, we adopted the declarative language `ocaml`[1] (v3.06) because not only it fulfills all the features highlighted before, but also offers:

- Mechanisms of modules parameterisation, which greatly enhances developing speed and code reuse;
- an efficient garbage collector for automatic memory management;
- sum types and a pattern matching mechanism that represent a main feature in declarative programming;

[1] Refer to `http://www.ocaml.org` for more information.

- native and efficient support for useful data structures such as polymorphic types;
- functional and imperative styles that can be mixed to get the best of the two approaches;
- cross-platform compilation.

Finally, `ocaml` is quite widespread and the current release comes out from years of development at INRIA[2]. In the following sections we highlight some aspects of `ocaml` which are a key mechanisms in $\mathcal{A}$SPASyA implementation.

Data types The language allows the definition of new types applying operators to primitive types[3] . For our purposes, the most important operators are cartesian product and disjoint union. Cartesian product allows the creation of tuples of values while disjoint union allows the definition of *sum types*. Indeed, a sum type is the disjoint union of sets of values where, to distinguish each set, a *constructor* is given, which not only constructs values of the new type, but also allows access to them by means of pattern matching.

Example 4.1 *Using sum types one can recursively define cIP messages:*

```
type  message =
      | SKey of string * int
      | PKey of string * int
      | Crypt of message * message
      | Var of string * int
      ...
```

A value of type message can be a symmetric key, identified by the constructor `SKey` *and parameterised with a cartesian product of a string (the key name) and an index relating the key with the principal owning it. Moreover a message can be a private key (*`PKey`*), a cryptogram (*`Crypt`*), a variable (*`Var`*) and so on, using a new constructor for every disjoint set of data. Values of type message are generated by applying constructors to data values. For example,* `SKey('k',1)`*, is the representation of the symmetric key k belonging to principal 1. Moreover,* `Crypt(Var('x',2), SKey('k',1))` *represents* $\{x_2\}_{k_1}$. $\diamond$

Data structures `ocaml` has a native support for lists. We use them as building blocks for defining trees, hash tables and incremental data structures with little programming effort and achieving high efficiency. Indeed, an `ocaml` list is implemented as a linked list so that, when a new element is added, there is no content duplication but a mere pointer redirection. Therefore using `ocaml` it is not necessary to code dynamic incremental structures by hand, because they are already present in the run-time support.

4.3 Representing configurations

The cIP semantics is given in terms of configuration reduction, so we recognised a configuration as the basic building block and implemented it first. A configuration is a triple $\langle \mathcal{C}, \chi, \kappa \rangle$ where $\mathcal{C}$ is a set of principal instances, χ a variable substitution and κ is the representation of the intruder knowledge about exchanged messages. Principal processes, variable substitutions and intruder knowledge heavily rely upon messages. The definition of messages as a sum type (Example 4.1) allows us to implement the action matching mechanism of the cIP semantics in terms of `ocaml` (efficient) pattern matching.

[2]INRIA is the French national institute for research in computer science and control. Refer to `http://inria.fr` for general information and to `http://cristal.inria.fr/` for `ocaml` specific information.

[3]Integers, floating-point numbers, characters, strings and boolean are predefined in `ocaml`.

Principal representation

Principals can be represented as sum types as well:

```
type  process =
        | Input of message * process
        | Output of message * process
        | Parallel of process * process
        | Sum of process * process
        | Nil
```

The definition of a process is straightforward and allows for the construction of parallel composition (`Parallel`) and non deterministic choice (`Sum`) of processes.

Not all constructable messages and processes have an associated semantics (e.g. cryptograms with non atomic keys) so we need to provide some methods to check for syntactic validity of an expression. More precisely we have to check that:

- Principals are closed;
- a variable must appear as bound only once in a process;
- each principal action must contain a valid message.

Some of this points are realised statically in the cIP parser, while others (as the correctness of messages) must be dynamically checked at run-time.

A *context* is represented as a list of principal instances.

Variable substitutions

Particular attention is required for representing variable substitutions. A substitution is a mapping from a set of variables to a set of messages. Substitution application to m changes the occurrences of variables in m according to the mapping. Substitutions are necessary in various cases:

- The intruder has to find a variable substitution to match principal actions,
- formula verification give rise to substitutions,
- substitutions have to be composed and are applied to configurations.

The representation of substitutions has been a crucial point in the realisation of $\mathcal{A}$SPASyA. It has been defined as a *functional type*[4] whose values can be hence applied to messages. The information about variable assignments is stored in a table; with this representation, composition of substitutions reduces to merging of tables (exploiting `ocaml` mechanisms for incremental structures). Given a message m its structure is analysed by recursive pattern matching and each time a variable is found, a table look-up is performed. When composing two substitutions, recent assignments are stored in the top of the table to reduce the list traversal time.

Intruder knowledge

The knowledge structure has to perform two main tasks:

(i) It represents what the intruder knows about a trace, and

(ii) is used to forge messages that must be sent to principals.

[4]A functional type is $t \to t'$ and represent functions from t to t'.

The knowledge of the intruder is naturally represented as a list of messages but to realise (ii) we need some more work. The intruder has the power to forge new messages from existing ones, either pairing or unpairing[5] them or encrypting cryptograms, or decrypting cryptograms when the correct key is known. Every message added to κ is decomposed in atomic messages (i.e. keys, names, cryptograms whose decrypting key is unknown) by repeated application of the unpairing/decrypting rule as follows:

```
add(m,κ)
 match m with
  key k:   κ = κ ∪ {k}; for each {l}_k ∈ κ do add(l,κ − {l}_k)
  name n:   κ = κ ∪ {n}
  pair (s,r):  add(s,κ); add(r,κ)
  {l}_k:  if k ∈ κ then add(l,κ) else κ = κ ∪ {m}
```

This computation is performed once for every added message and it is particularly useful to check for derivability of a message m. Indeed, in a so structured κ, derivability of m is reduced to checking membership of atomic components of m in κ. In pseudocode:

```
is-derivable(m,κ)
 match m with
  key k:k ∈ κ
  name n:n ∈ κ
  pair (s,r):is-derivable(s,κ) ∧ is-derivable(r,κ)
  {l}_k:(k ∈ κ ∧ is-derivable(l,κ)) ∨ m ∈ κ
```

An identification number (kId) is assigned to each instance of the knowledge and pointers to knowledges are stored in a hash table indexed by kId values. Knowledges can be accessed in constant time using kId as a search key. The usefulness of the hash table will be explained in section 5.1.

[5]This means that if the intruder knows a message m, he can forge an infinite set of messages. Indeed $(m, m, \ldots)$ is forgeable by repeated pairing.

Chapter 5

$\mathcal{A}$SPASyA: A Symbolic Model Checking Algorithm

$\mathcal{A}$SPASyA implements the reduction relation $\twoheadrightarrow$ prescribed by the cIP semantics, in two separated modules: `step` and `join`. The construction of the state space begins in the `join` module by adding different several instances of principals to the empty configuration. Next, `step` is in charge of generating the entire state space for every initial configuration built by the `join` module. Due to the definition of $\twoheadrightarrow$, the resulting state space is a tree of configurations and a trace is a path from a leaf to the root. $\mathcal{A}$SPASyA also implements a second aspect of the formal framework, namely, protocol properties. They are specified using $\mathcal{PL}$ logic and checked for satisfability against configurations. This is reflected in the architecture of $\mathcal{A}$SPASyA by the presence of the modules aimed at representing formulae (`logic`), handling verification (`verifier`) and resolting constraint systems (`csolver`).

The rest of the section describes the implementation of these modules, focusing on the model checking algorithms and their implementation.

5.1 Reduction steps

We start by considering the `step` module. Given a configuration $\langle \mathcal{C}, \chi, \kappa \rangle$, `step` is in charge of applying (*in*) and (*out*) rules of cIP semantics, to generate a new configuration. We can highlight three crucial points:

- Find principal actions ready to be fired,
- execute a principal action,
- generate a new configuration.

The first step is straightforward and realised with the analysis of the principal process structure. Every action to be fired is then processed according to the semantics given in Table 3.1. For an output action we have to consume the action, modify the context, and update the intruder knowledge with the sent message.

If we consider an input action $in(d)$, things become more difficult. The intruder can send a message m provided that a substitution γ such that $d\gamma \sim m$ and $\kappa \triangleright m$ exists. For every matching message we generate a new configuration, updating the context and the knowledge with the composition of the corresponding γ and χ. Difficulties arise because the intruder can generate a possibly infinite set of messages in response to a single input action. As an example consider $in(?x)$, the action of a principal waiting for a datum to store in x. Every possible message derivable from κ can be sent and, if κ is not empty, an infinite braching of possible configuration is introduced. Indeed, an infinite number of different substitutions (and hence configurations) can be generated. In order to face with this problem cIP has been equipped with a symbolic semantics. The formal tool for facing the infinite branching problem is constituted by *symbolic variables* (cfr. Chapter 3). The representation of symbolic variables is obtained by extending normal variables (see Example 4.1) with some parameters:

- An integer representing the identification number of κ;
- a field containing information about type. Indeed, the evolution of the computation will possibly impose restrictions on the set of values for a symbolic variable $x(\kappa)$. Restrictions are useful in trying to reduce the amount of generated substitutions for an action matching, and they will be described in section 5.2.

Example 5.1 *Symbolic variables are represented adding a new constructor to message definition (Example 4.1), namely* `SVar`*. Suppose we have the symbolic variable $x(\kappa)$ belonging to principal A_2 and $kId(\kappa) = 1$ whose internal representation is* `SVar('x',2,1,Gen)`*, where* `Gen` *is the "type" of data represented by x. Refer to Section 5.2 for details.* $\diamond$

Symbolic step

$\mathcal{A}$SPASyA implements a symbolic matching mechanism (introduced in [10, 41]) that, given a datum d and a knowledge κ , yields a symbolic substitution γ and a message m such that $\kappa \triangleright m$ and $m\gamma \sim d\gamma$. The calculation of a substitution γ for a datum d is based on a procedure defined by structural induction on data. The procedure introduced in [10, 41] handles many details of message matching. However, we are interested in giving a general idea of the mechanism, hence we show a simplified version. When the intruder has to derive a message matching d, some interesting cases arise:

1. d is a binding variable $?x$. This case is handled by the substitution $x \to x(\kappa)$ with κ representing the current knowledge.
2. d is a cryptogram $\{d'\}_k$ and the intruder knows the key needed to decrypt ($\kappa \triangleright k^{-1}$). The intruder can generate a message matching d if he is able to match d'.
3. d is a cryptogram $\{d'\}_k$ that the intruder can not decrypt. The intruder cannot know the structure of d' directly. It might be the case that one or more "potentially matching" cryptograms belong to κ, independently of whether or not the key k is in κ. The symbolic matching procedure scans κ for such cryptograms and, if possible, returns a matching substitution.
4. d is a cryptogram $\{d'\}_{x(\kappa_{old})}$. This case considers cryptograms where the encryption key is the symbolic variable $x(\kappa_{old})$ referring to a former intruder knowledge κ_{old}, that contains a finite number of keys. Hence the intruder can assign to $x(\kappa_{old})$ a value chosen among a finite number of possible values. The rest of the matching procedure is handled as in case (2)

Notice that the access to former knowledge is performed efficiently by a look-up in a hash table of knowledge pointers as specified in Section 4.3.

Example 5.2 *As an example consider the principal $A \triangleq ()[in(?x).in(\{na\}_x)]$ that waits for a datum and then for a cryptogram containing a nonce, encrypted with x. After that the intruder matches the first action, applying case (1), A is reduced to $[in(\{na\}_{x(\kappa)})]$. The intruder knows that $x(\kappa)$ is used as a key, so every generated substitution γ (finite in number) will contain an assignment such that $x(\kappa) \mapsto \lambda$, where λ is one of the keys in κ, as specified* *for (4). Suppose* $\kappa = \{k, I^+, I^-, na\}$ *then the symbolic matching procedure generates the three substitutions* $\gamma_1 = \{x(\kappa) \mapsto k\}, \gamma_2 = \{x(\kappa) \mapsto I^+\}, \gamma_3 = \{x(\kappa) \mapsto I^-\}$ *such that* $\{na\}_{x(\kappa)}\gamma_1 \sim k$, $\{na\}_{x(\kappa)}\gamma_2 \sim I^+$ *and* $\{na\}_{x(\kappa)}\gamma_3 \sim I^-$. $\diamond$

Symbolic matching is the core of the reduction step procedure which is reported in the following pseudocode:

```
single-step(⟨𝒞, χ, κ⟩)
 R = ∅
 A =set of processes in 𝒞 ready to fire an action
 for each a ∈ A do
  match a with
   in(d).P:
    Γ = symbolic-match(d,κ)
    for each γ ∈ Γ do
     R = R ∪ {⟨(C − {a}) ∪ Pγ, χγ, κ⟩}
   out(d).P:
    Γ = symbolic-match(d,κ)
    for each γ ∈ Γ do
     R = R ∪ {⟨(C − {a}) ∪ Pγ, χγ, κ ∪ dγ⟩}
 return R
```

The return value of `single-step` is the set of configurations resulting from one step of symbolic reduction. More precisely, R is a stack of configurations; `single-step` is invoked on every element in R to build the state space.

Configurations are heavily used in state space generation because, as we can see from `single-step`, every action of the intruder modifies the context. In general the reduction semantics always modifies the current configuration acting on the context, the variable substitution χ and the knowledge κ. Exploiting `ocaml` lists, each of this structures is implemented incrementally, storing only the differences between a configuration and the ones obtained from it.

5.2 Two optimisations

The iteration of `single-step` builds the whole state space. Even for small protocols, the state space generated is large due to the explosion caused by action interleavings. Pruning strategies are needed to shrink the state space as much as possible.

First, as proved in [10, 41], output of regular principals can be anticipated without any loss of significant traces. The intruder modeled is hence "eager" since he "learns" as much as possible from messages sent by regular principals. When all output messages have been collected, the intruder tries to generate messages for the waiting principals. This observation allows us to limit the dimension of state space without cutting off traces that lead to attacks. Intuitively, since the knowledge of the intruder increases monotonically, observing more data sent by the principals, can only increase the "power" of the intruder. Therefore, `single-step` has been modified in the implementation, to generate only those traces where output actions are fired in advance with respect to the input ones. More precisely, whenever we have to generate the successors of a configuration we check if it contains output actions. In such a case, we have only one successor which is the state obtained by performing all the output actions at once. Otherwise, we have a finite number of successors for each input action.

The second source of state explosion is the large number of possible substitutions generated by the intruder to match a single input action. For example, when the intruder knows that a symbolic variable $x(\kappa)$ is used as a key (Example 5.2), it generates a different substitutions for every key in κ. The situation becomes more dramatic when $x(\kappa)$ is used as a name, so a transition for every name in κ is performed. We extended the formal framework adding "type" information to symbolic variables in order to reduce the branching factor. For instance, if we know, by structural analysis of messages, that x stands for a message of type t, we substitute $x(\kappa)$ with $x^t(\kappa)$, where t ranges on $\{P, pb, pr, sy\}$ (respectively denoting 'principal names', 'public', 'private', 'symmetric').

Example 5.3 *Consider the principal* $A \triangleq ()[in(?x).in(\{na\}_x)]$ *of Example 5.2. Whenever the intruder has to forge a message matching with* $?x$*, a symbolic variable* $x(\kappa)$ *is sent. Afterwards, the second action imposes a constraint on* $x(\kappa)$ *because the intruder "learns" that* x *is intended to be a key, hence we consider only those transitions where a key in* κ *is substituted for* x*. All those transitions are represented by the substitutions* $x(\kappa) \to x^{pb}(\kappa)$*,* $x(\kappa) \to x^{pr}(\kappa)$ *and* $x(\kappa) \to x^{sy}(\kappa)$*. Figure 5.1 highlights the efficiency of such approach*

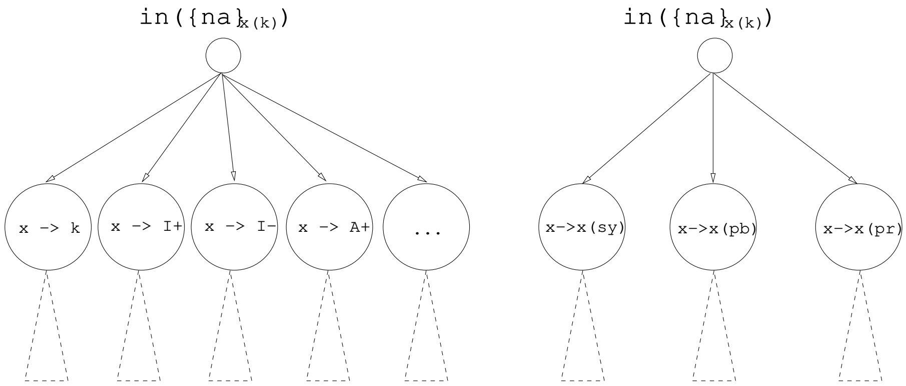

Figure 5.1: Optimised branching.

on a fragment of state space for the second action of A*. The leftmost tree does not use the optimisation and the branching factor depends on the number of keys in* κ*. The right tree has a branching factor dependent on the number of key* types. $\diamond$

5.3 Joining principals

Principal instances are organized into contexts. The only way for a principal to enter a context is by means of the join operation. The problem of verifying a protocol with an unbounded number of principals is undecidable so we let the user specify the upper bound m on the number of principal instances. Moreover we assume that a join operation is performed before any reduction step, on the empty initial configuration (i.e. the configuration made of the empty context, the undefined substitution and the initial intruder knowledge[1]). Given a protocol with n roles, the number of different initial contexts containing m instances that can be generated, is $\binom{m+n-1}{n}$. In cIP semantics the join operation is not trivial due to the presence of open variables. Every time a principal $A \triangleq (\tilde{v})[P]$ is going to be added, we have to:

- Inform the intruder about the presence of A;
- connect A to other principal by assigning values to their open variables.

The intruder adds to its knowledge the name of the joining principal A together with its public key A^+. All the principals join in a single step, so the intruder fills κ with all the

[1]The initial intruder knowledge always contains the intruder name I and its asymmetric keys I^+, I^-.

names at once. The gist of the join operation is the connection of open variables. An open variable can represent a principal name and hence its public key or a symmetric key shared with some other principal. The first step is to assign every open variable x representing a name to a symbolic variable $x^P(\kappa)$ representing all the names in κ. The second step is to assign a value to the remaining open variables. We have to generate a substitution for every possible assignment of open variables to key names. If the same value is assigned to two open variables, they will allow the corresponding principals to share keys. For instance, $(a)[\ldots]$ and $(b)[\ldots]$ can share a secret key k, if $a \mapsto k$ and $b \mapsto k$. Following this observation we can see that the substitutions are in one-to-one correspondence with the partitions of the open variable set.

Example 5.4 *Consider a context with three open variables, say a, b and c. There are five ways of assigning them:*

$$\begin{array}{lcl} a = b = c & \sim & \{a, b, c\} \\ a = b, c \neq a & \sim & \{a, b\}, \{c\} \\ a = c, b \neq a & \sim & \{a, c\}, \{b\} \\ b = c, a \neq b & \sim & \{b, c\}, \{a\} \\ a \neq b, b \neq c, c \neq a & \sim & \{a\}, \{b\}, \{c\} \end{array}$$

In the first case every one share the same key, while in the last case no key is shared. ◇

Let n be the number of roles of a protocol, m the number of instances of a configuration. Then, the number of initial configurations grows exponentially with m, so the dimension of the problem may become intractable even for small m. For every initial context we have to further connect the open variables, leading to a big number of starting configurations. However, let us remark that, in realistic protocols, $n \leq 5$.

5.3.1 Controlling the state explosion

The join mechanism is the basic building block to connect principals properly within protocol sessions. In general, the informal specification contains many implicit assumptions that play a fundamental role in the analysis and verification of the protocol. $\mathcal{A}$SPASyA lets the user to specify those assumptions by means of a property that has to be satisfied by every configuration. The join operation is equipped with a mechanism to check invariant properties on protocol principals: This is simply achieved by means of *Connection formulae*, that will be illustrated in Chapter 6. A connection formula is a $\mathcal{PL}$ formula that constraints the join operations.

Example 5.5 *Consider the WMF protocol specification reported from Example 3.1.*

$$\begin{array}{lcl} A & \triangleq & (x, s)[out(A, \{T^a, x, kab\}_s)], \\ S & \triangleq & (u, ak, v, bk)[in(u, \{?t, v, ?r\}_{ak}).out(\{T^s, u, r\}_{bk})], \\ B & \triangleq & (z)[in(\{?y, ?q, ?w\}_z)]. \end{array}$$

The following connection formula specifies the property that must hold in every trace:

$$\begin{array}{rl} \phi = & (\exists S.o : true) \wedge \forall S.l : \\ & (\exists A.i : (u_l = A_i \rightarrow ak_l = s_i) \wedge (v_l = A_i \rightarrow bk_l = s_i)) \wedge \\ & (\exists B.j : (u_l = B_j \rightarrow ak_l = z_j) \wedge (v_l = B_j \rightarrow bk_l = z_j)) \end{array}$$

Property ϕ states that at least a server must exist, and whenever an instance of A is connected to a instance of S as initiator ($u_l = A_i$) or responder ($v_l = A_i$), then they share a key ($ak_l = s_i$ or $bk_l = s_i$). The same happens for instances of B. ◇

After every initial configuration is generated by `join`, a formula verifier is invoked and every configuration not satisfying the connection is discarded and no further expanded.

Example 5.6 *For instance, referring to Example 5.5 consider the initial configuration* Γ *obtained with the substitution:*

$$\delta = \begin{cases} s, bk & \mapsto & kbs \\ z, ak & \mapsto & kas \\ v & \mapsto & B, \\ u & \mapsto & A \end{cases}$$

Connection formula ϕ *does not hold in* Γ *that will not be further expanded. Indeed,* δ *represents a principal connection that does not satisfy the hypothesis on the sharing of keys because principal* A *is connected to the server as initiator* ($u \mapsto A$) *but share a key with* S *as responder* ($s, bk \mapsto kbs$). $\diamond$

5.4 Representing formulae

Formulae are represented as a sum type encapsulating messages. As seen in Section 3.2, $\mathcal{PL}$ formulae express relations between messages, namely the equality ($=$) and the derivability ($\triangleright$) relations. Moreover $\mathcal{PL}$ logic, allows us to quantify over instances of principals. The `ocaml` representation of a formula follows:

```
type  formula =
       | Equal of message * message
       | Derive of message
       | Forall of index * string * formula
       | Exists of index * string * formula
       | And of formula * formula
       | Or of formula * formula
       | Not of formula
       | True | False
```

Type *index* is overloaded: It represents both a pointer to a symbol table storing indexes names, and a principal instance number. With this representation, operations on formulae are simplified. Consider the quantification constructors: They store enough information to solve every index reference contained in the quantified formula. For instance, the $\mathcal{PL}$ formula $\forall A.i : x_i = A_i \vee \not\triangleright na_i$ is represented as:

```
Forall(i,'A',Or(Equal(Var('x',i),Name('A',i)),Not(Derive(Nonce('na',i))))).
```

Before entering the verification process, formulae are normalised. Given a context $\mathcal{C}$and a formula ϕ, the normalisation of ϕ is a two-step process: The first step consists of expanding quantifiers with respect to principal instances i $\mathcal{C}$. More precisely the formula $\forall A.i : \phi$ is expanded as:

$$\phi[^{j_1}/_i] \wedge \phi[^{j_2}/_i] \wedge ...\phi[^{j_n}/_i] \quad with \quad j_m \in \{j | A_j \in \mathcal{C}\}$$

The second step is the transformation of an expanded formula to a disjunction of conjuncts, by means of the usual De Morgan laws.

Example 5.7 *Let* $\mathcal{C} = \{A_1, A_2, B_3\}$ *and* $\phi = \forall B.j : \exists A.i : x_j = y_i \wedge y_i \neq B_j$. *After quantifiers expansion we get* $\phi = (x_3 = y_1 \wedge y_1 \neq B_3) \vee (x_3 = y_2 \wedge y_2 \neq B_3)$ *where* i *and* j *are substituted with the actual indexes.* ϕ *is already normalised, being a disjunct of conjuncts, and is ready to be verified.* $\diamond$

5.5 Constraint systems

We exploit a constraint system solver for model checking $\mathcal{PL}$ formulae. Models for $\mathcal{PL}$ formulae are configurations that may contain symbolic variables. We remark that checking for satisfability of a formula may give rise to an infinite number of possible substitutions (as in symbolic matching, Section 5.1). For instance, consider the formula ϕ in Example 5.7; ϕ is made of positive ($x_3 = y_1$ and $x_3 = y_2$) and negative ($y_1 \neq B_3$ and $y_2 \neq B_3$) atoms. Suppose that ϕ is going to be checked on a configuration $\langle \mathcal{C}, \chi, \kappa \rangle$, where $\chi = \{x_3 \mapsto x_3(\kappa), y_1 \mapsto y_1(\kappa)\}$. We have to substitute the variables in ϕ as dictated by χ and then check for satisfability of each disjunct which depends on satisfability of equalities and inequalities. Indeed, in order to check for the satisfability of $x_3(\kappa) = y_1(\kappa)$ one should prove that there exists a message m ($\kappa \triangleright m$) and a substitution γ such that $x_3(\kappa)\gamma = y_1(\kappa)\gamma = m$. Since the number of such subsitution is infinite (one for every message derivable from κ), a symbolic constraint solver is used to deal with such situations.

Positive atoms are dealt with by means of standard unification and give rise to constraints easily calculated with a procedure similar to symbolic matching. For example $x_3(\kappa) = y_1(\kappa)$ gives rise to the assignment $x_3 \mapsto y_1$. More complex matching may generate more than one substitution, but always finite in number thanks to symbolic techniques.

Negative atoms are always true if the knowledge is not empty because the intruder can always derive two different messages. For example the negative literal $y_1(\kappa) \neq B_3$ of $\phi\chi$ holds for any substitution that maps y_1 on a name different from B_3. To finitely represent this situation the negative assignment $\{y_1 \not\mapsto B_3\}$ is generated in order to record this constraint.

Let Q be the constraint store. As seen, a constraint is a variable assignment which can be positive ($x \mapsto d$) or negative ($x \not\mapsto d$). A constraint store Q is valid whenever:

(i) $(x \mapsto d) \in Q \wedge (x \mapsto d') \in Q \Rightarrow d \neq d'$

(ii) $(x \mapsto d) \in Q \iff (x \not\mapsto d) \notin Q$

Condition (i) states that a variable must be assigned to at most one value. Condition (ii) requires that a value for a variable must be chosen without violating any negative constraint present in Q.

The constraint solver is implemented by checking that (i) and (ii) are satisfied after each new constraint is added to Q by the verification process.

5.6 Formula satisfaction

Once a formula is normalised, it is transformed in $\phi = \bigvee_{i=0}^{n} \psi_i$ where each ψ_i is made only of conjunctions of atoms. Checking for the satisfability of a normalised formula means finding a substitution γ such that $\psi_i\gamma$ is true for some i. If ψ_i can not be satisfied for any i then ϕ is unsatisfable. The verification process relies on symbolic matching functions and the constraint solver to achieve its goal. In pseudocode:

```
check(ψ)
  let ψ = φ1 ∨ φ2... ∨ φn;
  for each φi = ρ1 ∧ ρ2... ∧ ρm do
   Q = ∅;
   for each ρj do
     γj = solve(ρj);
     Q = Q ∪ γj;
  if correct(Q) return true
 return false;
```

Here *solve* returns a set of positive or negative constraints depending on ρ_j, and *correct* checks for the coherence of the constraint store Q. This pseudocode can be easily modified to return also the substitutions that make ψ true. Moreover, *solve* may return more than one γ for a single atom, so the verifier generates non-deterministically all the possible substitutions (which are symbolic and hence finite). Non determinism is simulated with a stack of substitutions that allows backtracking when reaching a non satisfable set of constraints.

5.7 Mixing all together

The problem of protocol verification with finite sessions was shown to be NP-complete in [37], so much effort has been spent in trying to achieve tractability for a reasonable input size (i.e. protocol complexity and number of instances). Searching for an attack means to generate the state space and check the property. This is a quite general algorithm which we tuned in several ways to achieve better performance and flexibility.

As noticed in Section 5.3.1, the information contained in connection formulae may be exploited to guide the expansion of the state space and restrict the search. Due to the modular architecture of $\mathcal{A}$SPASyA this is easily accomplished by letting `join` to interact with `verifier`. Every generated initial configuration is passed to the verifier and checked against the corresponding connection formula. If the test is successful the configuration is forwarded to `step`.

Upon reaching a terminal trace, protocol properties need to be checked: When the security formula is not satisfied and the connection formula holds, an attack trace is found. Again, this is accomplished by the interaction between `step` and `verifier`. The rest of the section will recap the details of the verification algorithm.

We can highlight four steps of computation also depicted in Figure 5.2:

1. As said, state space can be seen as a tree of configurations. Initially, the root node contains the configuration made of the empty context, the undefined substitution $\perp$ and the initial intruder knowledge κ_0.

2. The next step is the construction of the initial contexts; each initial context contains a number of principal instances m (specified by the user). Then, we normalise each $\mathcal{PL}$ formulae (connection and security properties) according to each initial configuration.

3. At third step, we are able to execute the join operation generating all the possible connections between open variables. The state space is already quite large, so we prune it for the first time, checking the satisfability of the invariant property. If a configuration does not satisfy the invariant, it will never give rise to an attack trace and can hence be pruned.

4. After the pruning, each trace t is expanded until no more actions can be performed. If t is a terminal trace (i.e. all principals have performed all their actions) we check for the satisfability of the security property.

If the security property does not hold in the terminal configuration of some trace , then we cannot claim that we found an attack trace, because the corresponding substitution may not satisfy the invariant property. Therefore, we perform a fast check on the invariant to assess the validity of the attack. Notice that, while step 3 checks whether a feasible substitution γ might exist or not, here a concrete substitution is applied and checked. It may be the case that, during trace generation, we refined a value of a symbolic variable that was assigned during the join operation. We have to be sure that the value assigned along the trace is the same of the value specified in γ. When an attack trace is found, it is used to reconstruct the cIP process of the intruder. There could be more than one substitution that

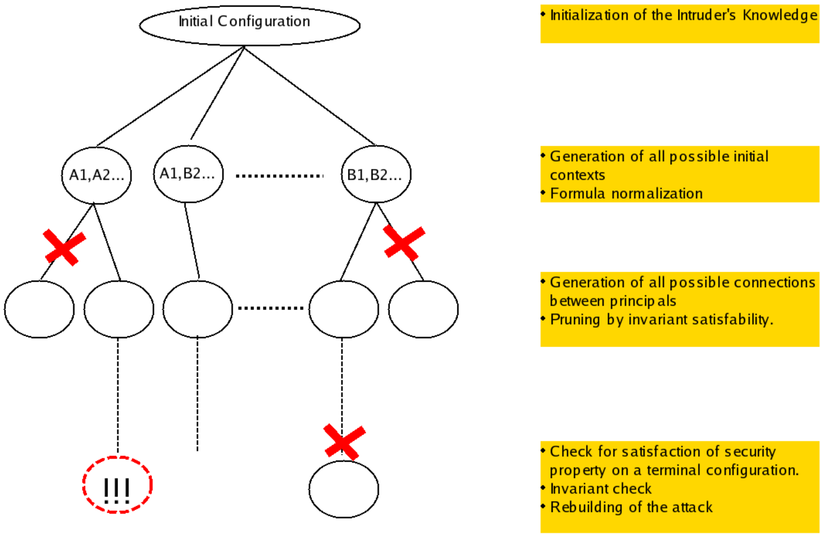

Figure 5.2: A graphical representation of the verification algorithm

allows the attack for the same trace. Hence, $\mathcal{A}$SPASyA let the user to specify how many attacks must be reported for the same trace.

We apply the verification algorithm to the WMF protocol in Example 3.1, to clarify each step. We simulate the algorithm for $m = 3$ and highlight interesting points.

1. The root configuration is created from the initial knowledge specified by the user. Suppose $\kappa_0 = \{I, I^+, I^-\}$, hence $\langle \emptyset, \perp, \kappa_o \rangle$ is the initial configuration.

2. Principals join the initial context giving rise to 10 different contexts. Two of the generated contexts are $\mathcal{C}_1 = \{A_1, B_2, S_3\}$ and $\mathcal{C}_2 = \{A_1, A_2, A_3\}$. The corresponding configurations contain knowledges where the names of the participants have been added, respectively $\kappa_1 = \kappa_0 \cup \{A_1, A_1^+, B_2, B_2^+, S_3, S_3^+\}$ and $\kappa_2 = \kappa_0 \cup \{A_1, A_1^+, A_2, A_2^+, A_3, A_3^+\}$. Suppose we have the security property ψ of Example 3.5 and the connection formula ϕ of Example 5.5:

$$\begin{aligned}\psi = \quad & \forall B.j : \exists S.l : \exists A.i : \\ & (v_l = B_j \land \; u_l = A_i \land \; x_i = B_j) \rightarrow \\ & (t_l = T_i^a \land \; y_j = T_l^s \land \; w_j = kab_i \land q_j = A_i)\end{aligned}$$

$$\begin{aligned}\phi = \quad & (\exists S.o : true) \land \forall S.l : \\ & (\exists A.i : (u_l = A_i \rightarrow ak_l = s_i) \land (v_l = A_i \rightarrow bk_l = s_i)) \land \\ & (\exists B.j : (u_l = B_j \rightarrow ak_l = z_j) \land (v_l = B_j \rightarrow bk_l = z_j))\end{aligned}$$

We normalise[2] ψ and ϕ with respect to each $\mathcal{C}_i$ obtaining the corresponding ψ_i and ϕ_i:

$$\begin{aligned}\psi_1 = \quad & v_3 = B_2 \land \; u_3 = A_1 \land \; x_1 = B_2 \rightarrow \\ & t_3 = T_1^a \land \; y_2 = T_3^s \land \; w_2 = kab_1 \land q_2 = A_1\end{aligned}$$

$$\begin{aligned}\phi_1 = \quad & true \land (u_3 = A_1 \rightarrow ak_3 = s_1) \land (v_3 = A_1 \rightarrow bk_3 = s_1) \land \\ & (u_3 = B_2 \rightarrow ak_3 = z_2) \land (v_3 = B_2 \rightarrow bk_3 = z_2).\end{aligned}$$

Differently, the context $\mathcal{C}_2$ gives rise to the unsatisfable formula $\phi_2 = false \land true$ because $\mathcal{C}_2$ does not contain any instance of S. Hence the existential and universal quantification in ϕ are nomalised to $false$ and $true$ respectively.

[2] For the sake of readability, formulae are only partially normalised.

3. For each context, all possible assignments of open variables are generated and the corresponding configurations constructed. One of the substitutions generated for $\mathcal{C}_1$ is

$$\gamma_1 = \begin{cases} x_1 & \mapsto & x_1^P(\kappa_1) \\ u_3 & \mapsto & u_3^P(\kappa_1) \\ v_3 & \mapsto & v_3^P(\kappa_1) \\ ak_3, s_1, bk_3, z_2 & \mapsto & k. \end{cases}$$

Indeed, the first join phase assigns every open variable representing a name (x_1, u_3, v_3) to a typed symbolic variable, whereas the second phase creates links between remaining variables assigning them to key names. Formula ϕ_1 is then checked on $\langle \mathcal{C}_1, \gamma_1, \kappa_1 \rangle$, and holds for at least an assignment, namely $\{x_1 \mapsto A_1, u_3 \mapsto A_1, v_3 \mapsto B_2\}$. Context $\mathcal{C}_1$ passes the invariant check while $\mathcal{C}_2$ is discarded because ϕ_2 never holds.

4. Symbolic matching procedure is repeatedly applied to configurations to obtain traces. Starting from $\langle \mathcal{C}_1, \gamma_1, \kappa_1 \rangle$ there is only one principal action that the eager trace generation procedure can select, namely the output action $out(A_1, \{T_1^a, x_1^P(\kappa_1), kab_1\}_k)$ of A_1. The sent message is added to κ_1 obtaining $\kappa_2 = \kappa_1 \cup \{T^a, x_1^P(\kappa_1), kab\}_k$, since the intruder does not know k and cannot open the cryptogram (refer to Section 4.3). Context $\mathcal{C}_1$ is modified and now contains A_1 represented by the *nil* process whereas B_2 and S_3 are not modified. The resulting configuration is $\langle \mathcal{C}', \gamma_1, \kappa_2 \rangle$ where

$$\begin{aligned} \mathcal{C}' = \{ \quad & A_1 : ()[nil], \\ & B_2 : ()[in(\{?y_2, ?q_2, ?w_2\}_k)], \\ & S_3 : ()[in(u_3^P(\kappa_1), \{?t_3, v_3^P(\kappa_1), ?r_3\}_k).out(\{T_3^s, u_3^P(\kappa_!), r_3\}_k)]\}. \end{aligned}$$

The successive application of the step procedure gives rise to two different possibilities: Selecting the input action of B_2 or the input action of S_3. We consider the former case where the intruder generates a matching substitution $\delta_1 = \{y_2 \mapsto T_1^a, q_2 \mapsto x_1^P(\kappa_1), w_2 \mapsto kab_1\}$; it represents the forwarding to B_2 of the cryptogram already sent by A_1. As before, the new configuration $\langle \mathcal{C}'', \gamma_1\delta_1, \kappa_2 \rangle$ is created with an updated context, the composition of the previously generated substitutions and the unmodified knowledge κ_2. We skip the next two steps, corresponding to the execution of the actions of the server S_3. The terminal configuration is:

$$\begin{aligned} \tau = \quad & \langle \{A_1 : ()[nil], B_2 : ()[nil], S_3 : ()[nil]\}, \\ & \gamma_1; \delta_1[{}^{T_1^a, kab_1, B_2}/_{t_3, r_3, v_3,}], \\ & \kappa_2 \cup \{T_3^s, u_3^P(\kappa_1), kab_1\}_k \rangle \end{aligned}$$

Formula ψ_1 is checked on τ and the concrete substitution $\delta = \{u_3 \mapsto A_1, x_1, q_2 \mapsto B_2, v_3 \mapsto B_2\}$ is found such that $\psi_1\delta$ does not hold. Indeed, in the antecedent of $\psi_1\delta$ $(v_3 = B_2 \wedge \ u_3 = A_1 \wedge \ x_1 = B_2)$ each atom holds whereas the consequent is false. Applying δ to the consequent of psi_1 we obtain from the atom $q_2 = A_1$ the corresponding atom $B_2 = A_1$ that is clearly false. Before claiming an attack, we have to perform an invariant check to be sure that values assigned to symbolic variables u_3, x_1 and v_3 do not violate the connection property. Indeed, $\phi_1\delta$ holds on τ since every implication can be satisfied by $\delta; \gamma_1$. and we can report an attack. Notice that this attack is caused by the sharing of the same key between A_1, B_2 and S_3. Modifying ϕ we can exclude such situations from the search.

5.8 Performance issues

In this section we briefly analyse efficiency of $\mathcal{A}$SPASyA's algorithm, considering both memory usage and computation time.

One of the main problems related to the handling of (very large) transition systems is memory consumption. These problems have been considered by other authors, see [42, 8, 9]. Notice that since we are interested in checking a formula when we reach a terminal trace, it is not necessary to maintain in memory all the state space but only the current trace. This has been accomplished using a (stack implemented) depth-first search strategy that coupled with the efficient `ocaml` garbage collector enables us to use a little amount of memory. For instance, a few space (around 3 Mb) is needed to store a trace in most of the protocol we have tested in Chapter 7. This also has impact on time efficiency because we gain speed for the lack of page swapping.

Computation time depends on the size of the state space and by the complexity of formulae to be verified. We reduced the dimension of the generated state space by several optimisations reported in Sections 5.2 and 5.3.1. Moreover $\mathcal{A}$SPASyA offer the user the possibility to search small and interesting portions of the state space by invariant pruning. For instance, as reported in Chapter 7, the verification of the Needham Schroeder protocol with four principal instances and with no restrictions on the connections, yields a state space of 374.905 configurations whereas restricting the search to consider contexts with two instances for each role, yields a state space of only 18.385 configurations.

Formula satisfaction is also a time consuming task, especially when symbolic variables are involved. There might be more than one variable assignment for a single symbolic trace that leads to an attack and reporting them all is very inefficient. $\mathcal{A}$SPASyA can report the full set of attacks or only the first attack found in a trace, allowing the user to perform a fast but less accurate search that can be subsequently refined. Refer to Chapter 7 for quantitative results on performed tests.

Chapter 6

$\mathcal{A}$SPASyA: A Verification Methodology

This chapter discusses how $\mathcal{A}$SPASyA can be used to verify security protocols. In this chapter we how our approach makes the verification process easier. Indeed, the user can build the whole specification in successive steps and can reduce as much as desired the search space. Moreover, the formalisation of an informal specification is a difficult and error prone process, and the presented methodology is useful to discover and avoid many of such errors.

Figure 6.1 represents the main phases of the $\mathcal{A}$SPASyA verification methodology. Verifying a protocol is based on a four-step procedure:

(i) Each role is formalised by a cIP principal,

(ii) the security property is specified by means of a $\mathcal{PL}$ formula,

(iii) conditions on connections are specified by means of a $\mathcal{PL}$ formula and,

(iv) initial knowledge is provided.

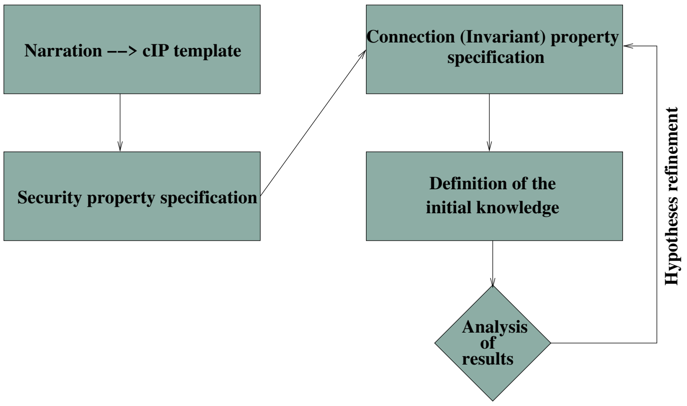

Figure 6.1: A graphical representation of the verification methodology

The basic idea of step (i) is to provide a cIP process for each principal corresponding to a role in the protocol. At each step, its behaviour depends on whether the principal is sending or receiving a message. The protocol analyser must define a cIP template specifying open variables of principals. Even though it seems that the formalisation of the narrative informal specification into cIP processes can hardly be done in a fully automatic manner, we can give some rule of thumbs:

- Initiator usually needs an open variable which the responder should join;
- if the identity of the partner is acquired in a communication, then no open variable should be necessary about that partner (unless checks on the identity are required);
- an open variable might be necessary when a principal must interact with a server, to share a secret (e.g. a key or a name).
- Each step of the informal specification corresponds to an input action for the receiver and an output action for the sender. Each time a principal receives a name it is stored in a binding variable.

In step (*ii*) the property to be verified is defined in $\mathcal{PL}$. Security properties formalisation is a complex task and requires experience. $\mathcal{PL}$ logic helps by allowing the formalisation of:

- The impossibility for the intruder to know a particular datum d in a run of the protocol, when checking for secrecy properties,
- the relations between variables of different principals that must hold in every run, when more complex properties, like authentication, must be verified.

Step (*iii*) let the analyser specify a connection property, obtaining a double purpose:

- Specify the assumptions on the keys shared between principals, and
- prune the state space allowing for fast search on a reduced set of selected trace.

It is often the case that the result of a search (i.e. an attack trace) reveals that some assumptions on the protocol are not correctly formalised. By refining the hypothesis on principal connections, incorrect attacks can be filtered out (refer to Section 6.1.4 for an example).

Finally, step (*iv*) specifies the initial knowledge of the intruder to test a protocol under weaker conditions. It is used mainly for two purposes:

- Let the intruder know some secrets (e.g. compromised keys);
- Let the intruder know something about past interactions between principals (cryptograms exchanged in previous sessions).

The latter is especially useful in finding *replay attacks* where the intruder exploits messages appeared in previous session.

Steps (*iii*) and (*iv*) can be iterated in order to tune the connection conditions (and the initial knowledge), according to the results obtained in previous iterations.

6.1 Finding attacks with $\mathcal{A}$SPASyA

We highlight the effectiveness of our methodology by analysing real protocols. We therefore introduce two well-known protocols, namely asymmetric Needham-Schroeder [33] and KSL [25], and use them as running examples.

6.1.1 Needham-Schroeder protocol

The asymmetric Needham-Schroeder protocol is based on public key cryptography and aims at guaranteeing mutual authentication between two principals, A and B, which communicate with a trusted third party S (certification authority) for the distribution of public keys. The narration description is:

$$
\begin{array}{lll}
(1) & A \rightarrow S: & (A, B) \\
(2) & S \rightarrow A: & \{B^+, B\}_{S^-} \\
\mathbf{(3)} & A \rightarrow B: & \{A, na\}_{B^+} \\
(4) & B \rightarrow S: & (B, A) \\
(5) & S \rightarrow B: & \{A^+, A\}_{S^-} \\
\mathbf{(6)} & B \rightarrow A: & \{na, nb\}_{A^+} \\
\mathbf{(7)} & A \rightarrow B: & \{nb\}_{B^+}
\end{array}
$$

The trusted server S stores public keys and distributes them upon request, encrypted with its own private key. Messages (1), (2), (4) and (5) are used for this purpose while the rest of the protocol reaches the mutual authentication goal. In our analysis we consider S trusted, namely, we assume that the keys are correctly delivered by S, and therefore we consider a usual "brief" version of the Needham-Schroeder protocol given by messages (3), (6) and (7). Principal A sends a cryptogram containing a nonce, na, and its own identity, encrypted with the public key of B; B replies with a message containing na and a newly generated nonce nb, authenticating itself with A. Indeed, when A decrypts the message in (6), can deduce that B is alive and communicating since only B is able to decrypt message (1) and acquire na. The last message achieves mutual authentication; A sends nb back to B, letting B conclude that A (the only principal able to decrypt (2)) is running and communicating. Even if the above informal explanation is quite convincing (and it has been for long time), the protocol is flawed. A first non trivial flaw is the well known Lowe attack [27], while a second attack will be described at the end of this section.

6.1.2 Verifying the Needham-Schroeder protocol

The cIP formalisation of the protocol is given below:

$$
\begin{array}{lll}
A & \triangleq & (y)[out(\{A, na\}_{y^+}).in(\{na, ?u\}_{A^-}).out(\{u\}_{y^+})] \\
B & \triangleq & ()[in(\{?z, ?x\}_{B^-}).out(\{x, nb\}_{z^+}).in(\{nb\}_{B^-})].
\end{array}
$$

Notice that open variable y is intended for the identity of the partner of A.

In order to formalise the authentication property, we have to state that the intruder can never let B believe that he is communicating with A when it is not. This desirable property can be formalised by means of the following $\mathcal{PL}$ formula:

$$\psi_{NS} = \forall B.j : \exists A.k : z_j = A_k \rightarrow y_k = B_j.$$

which states that, for each instance B_j, if B_j believes that he communicated with an instance of A, whose name is received in the first input action of B ($z_j = A_k$), then that instance A_k was actually intended to communicate with B_j ($y_k = B_j$).

For simplicity we perform the search, without imposing any constraint on connections and assuming the standard initial knowledge. We consider contexts with two principal instances. $\mathcal{A}$SPASyA returns eleven attacks that violate the security property above: The first one is the well-known Lowe attack [28], while others have to be analysed carefully. Most of them are permutations of the same intruder actions. Six of them can be described as follows:

$$
\begin{array}{lll}
(1) & I \rightarrow B_1: & \{I, x_1(\kappa)\}_{B_1^+} \\
(2) & B_1 \rightarrow I: & \{x_1(\kappa), nb_1\}_{I^+} \\
(3) & I \rightarrow B_1: & \{nb_1\}_{B_1^+} \\
(4) & I \rightarrow B_2: & \{I, x_2(\kappa)\}_{B_2^+} \\
(5) & B_2 \rightarrow I: & \{x_2(\kappa), nb_2\}_{I^+} \\
(6) & I \rightarrow B_2: & \{nb_2\}_{B_2^+}
\end{array}
$$

where the symbolic variables $x_1(\kappa)$ and $x_2(\kappa)$ can assume any value derivable from κ. This class of intruders cannot be claimed as a proper attack, indeed the intruder is behaving as a "honest" instance of A and hence ϕ_{NS} cannot be satisfied by any instance of A which intended to communicate with I ($y_k = \mathbf{I}$). This, and others that one may encounter, not significant result can be easily filtered out, following the outlined methodology, by further specifying the security property. The following $\mathcal{PL}$ formula rules out the possibility that the intruder may directly communicate with instances of A and B:

$$\begin{aligned}\psi'_{NS} = \;& \forall A.i : na_i \in \mathfrak{K} \rightarrow x_i = \mathbf{I} \\ & \wedge \\ & \forall B.j : (nb_j \in \mathfrak{K} \rightarrow z_j = \mathbf{I}) \vee (\exists A.k : z_j = A_k \rightarrow y_k = B_j).\end{aligned}$$

The added conditions state that the nonces na generated by instances of A must remain secret, unless they were intended for I. Moreover the intruder can "legally" obtain nb_j if he initiated the protocol session with B_j as responder ($z_j = \mathbf{I}$).

Repeating the search, $\mathcal{A}$SPASyA reports five attacks; four of them are the permutation of a type flaw attack of which we give the informal description. Since no constraints are imposed to connections, the context with only two instances of B (B_1 and B_2) is generated. A possible trace of this context is:

$$\begin{array}{lll}(1) & I \rightarrow B_1 & : \{I, B_2\}_{B_1^+} \\ (2) & B_1 \rightarrow I & : \{I, nb_1\}_{B_2^+} \\ (3) & I \rightarrow B_2 & : \{I, nb_1\}_{B_2^+} \\ (4) & B_2 \rightarrow I & : \{nb_1, nb_2\}_{I^+} \\ (5) & I \rightarrow B_2 & : \{nb_2\}_{B_2^+} \\ (6) & I \rightarrow B_1 & : \{nb_1\}_{B_1^+}\end{array}$$

where the intruder interleaves two sessions with B_1 and B_2 and plays as initiator. Step (1) contains a type flaw, since the intruder exploits its own identity I, as a nonce. Hence, B_1 replies to the nonce challenge with message (2), that I forwards to B_2 to make him generate message (4). The intruder can now decrypt message (4), therefore secrecy of nb_1 is violated ($nb_1 \in \mathfrak{K}$) even though $z_1 \neq \mathbf{I}$. At the end of the session (step (6)), B_1 concludes that B_2 initiated a protocol session and that B_2 intended to communicate with B_1, while B_2 has actually initiated the protocol with the intruder.

6.1.3 KSL

KSL has been presented in [25] as an improvement of Kerberos [26] because it does not rely upon synchronized clocks to guarantee freshness of messages. It is based on private key cryptography and its goal is repeated mutual authentication between two principals, A and B, communicating with a trusted third party S. The protocol is divided in two parts: An initial message exchange to establish a session key between principals, followed by the repeated authentication part. Repeated authentication is performed by means of an expiring ticket generated by B for A. Until the ticket is valid (not expired), A can re-authenticate itself with B without requesting a new session key from S. The informal specification is:

$$\begin{array}{lll}(1) & A \rightarrow B : & na, A \\ (2) & B \rightarrow S : & na, A, nb, B \\ (3) & S \rightarrow B : & \{nb, A, k^{ab}\}_{k^{bs}}, \{na, B, k^{ab}\}_{k^{as}} \\ (4) & B \rightarrow A : & \{na, B, k^{ab}\}_{k^{as}}, \{Tb, A, k^{ab}\}_{k^{bb}}, nc, \{na\}_{k^{ab}} \\ (5) & A \rightarrow B : & \{nc\}_{k^{ab}} \\ & & \\ (6) & A \rightarrow B : & ma, \{Tb, A, k^{ab}\}_{k^{bb}} \\ (7) & B \rightarrow A : & mb, \{ma\}_{k^{ab}} \\ (8) & A \rightarrow B : & \{mb\}_{k^{ab}}\end{array}$$

Messages (1) to (5) are the key exchange part whereas messages (6) to (8) are the repeated authentication. Server S shares a symmetric secret key with each principal. A generates a nonce na, and sends it to B starting the protocol. In message (2) B requests a new session key from S, which generates k^{ab} and sends message (3) using k^{as} (shared only by A and S) and k^{bs} (shared only by B and S). B decrypts the first cryptogram in message (3) and checks nonce nb and identity A. If the check is passed, B assume that k^{ab} is a fresh session key. Message (4) is crucial; B forwards $\{na, B, k^{ab}\}_{k^{as}}$ to A. Notice that, at this point, A and B share the session key. The second cryptogram of (4) is the ticket created by B to perform repeated authentication. It contains a *generalized time-stamp* [1], Tb, the session key, A identity and it is encrypted with a key k^{bb} known only by B. With $\{na\}_{k^{ab}}$, B reassures A of being alive and with nc asks A to do the same. Message (5) concludes the first part of KSL with A sending back nc crypted with the session key and achieving mutual authentication.

Principal A knowing k^{ab} and the ticket issued by B can re-authenticate itself performing steps (6) to (8). In message (6) B receives a nonce, ma, and a ticket B has previously generated. If the ticket is valid, B sends the encryption of ma to A together with a nonce, mb, used to assure A identity in message (8).

6.1.4 Verifying KSL key exchange part

In this section we apply the verification methodology to the key exchange part of KSL, emphasizing the use of connection formulae explained in Section 5.3.1

We start by giving the corresponding cIP template for A:

$$\begin{array}{lcl} A & \triangleq & (b, sk)[out(na, A). \\ & & in(\{na, b, ?r\}_{sk}, ?tkb, ?bn, \{na\}_r). \\ & & out(\{bn\}_r)]. \end{array}$$

It is important to emphasize the use of open variables. Variable b is used to store the identity of the partner of A. Notice that b appears in the second action of A and is used to check (via pattern matching) the identity of the partner. Variable sk is intended to store the key shared with S. The other data acquired by means of communications are stored in bound variables. The rest of KSL specification follows:

$$\begin{array}{lcl} S & \triangleq & (a, ak, b, bk)[in(?cna, a, ?cnb, b)). \\ & & out(\{cnb, a, kab\}_{bk}, \{cna, b, kab\}_{ak})] \\ \\ B & \triangleq & (sk)[in(?cn, ?u). \\ & & out(cn, u, nb, B). \\ & & in(\{nb, u, ?r\}_{sk}, ?tka). \\ & & out(tka, \{nt, u, r\}_{kbb}, nc, \{cn\}_r). \\ & & in(\{nc\}_r)]. \end{array}$$

It is important to remark that variables are local to a principal. When principal instances join a session of a protocol are associated with a unique index, so that there is no confusion between variables with the same name belonging to different principal instances (e.g. sk). Notice that open variables of S are used to let a principal connect and share a secret with S. More precisely a stores the identity of the initiator and ak holds the shared key. Variables b and bk are for the responder. Principal B has an interesting feature: It creates a ticket encrypted with a key known only to B. In cIP this is accomplished by assigning an explicit

[1] A generalized time-stamp is made of a current time-stamp of the local clock of B, an indication of lifetime and an "epoch" identifier to protect B against replay attacks. Refer to [34] for problems related with time-stamps.

name to the key (kbb), which is different from values of any open variable, and therefore cannot be shared.

KSL tries to achieve mutual authentication. Informally, this means that every time A wants to authenticate himself to B and they are correctly connected with a server S, data have to be exchanged correctly. Here a "correct" exchange means that a datum sent by a principal is received by the designated receiver without modifications.

$$\begin{aligned}\psi_{KSL} = \quad & \forall A.j : \exists S.i : \exists B.l : (b_j = B_l \wedge b_i = B_l \wedge a_i = A_j) \rightarrow \\ & (cn_l = na_j \wedge u_l = A_j \wedge bn_j = nc_l \wedge \\ & r_j = kab_i \wedge r_l = kab_i \wedge cna_i = na_j \wedge cnb_i = nb_l)\end{aligned}$$

Formula ψ_{KSL} is built from several literals: $b_j = B_l$ states the willingness of A_j to authenticate to B_l, while $b_i = B_l$ and $a_i = A_j$ (variables of S), refer to the correct connection of A_j and B_l to S_i. Once that these connections are given, literals on the consequent check correct data exchange:

- $cnb_i = nb_l$ and $cna_i = na_j$ specify the exchange of nonces between the principals and the server;
- $bn_j = nc_l$ and $cn_l = na_j$ are used to ensure the correctness of the nonce exchange between A_j and B_l;
- $r_j = kab_i$ and $r_l = kab_i$ state that session keys received by A and B must be the same and generated by the server, according to index i.
- $u_l = A_j$ specifies that B_l concludes A_j is communicating.

KSL assumes that keys are shared between the server and the principals: A connection formula is needed to state those assumptions. A possible invariant property is that in every run, a principal must be correctly connected to S. This amounts to say that every time A (B) is connected as initiator (responder) to S they should share a private key. In $\mathcal{PL}$

$$\phi_{KSL} = \forall S.i : \quad \exists A.j : (a_i = A_j \rightarrow sk_j = ak_i) \wedge \exists B.l : (b_i = B_l \rightarrow sk_l = bk_i)$$

According to this formula, for every server instance S_i, we require a correspondence between the name of the initiator (responder) and the open variable holding the shared key.

Running $\mathcal{A}$SPASyAwith three instances per context, no attack is reported given ψ_{KSL} and ϕ_{KSL}.

We might want to verify KSL augmenting the intruder power. Indeed, the previous intruder is limited because he cannot connect to a server as a "normal" principal (ϕ_{KSL} neglects this possibility). The protocol analyser may add the shared key to the initial knowledge making it available to the intruder. As for standard principals, sharing of secrets between the intruder and principal instances is performed by means of the join mechanism on the intruder open variables. Each intruder open variable is placed in the initial knowledge and will be used as a placeholder for the secret determined by the join. With this device we can check KSL, modifying ϕ_{KSL} to encapsulate this assumption:

$$\begin{aligned}\phi'_{KSL} = \forall S.i : \quad & (a_i = \mathbf{I} \rightarrow ak_i = sk_I) \wedge (b_i = \mathbf{I} \rightarrow bk_i = sk_I) \wedge \\ & \exists A.j : (a_i = A_j \rightarrow sk_j = ak_i) \wedge \exists B.l : (b_i = B_l \rightarrow sk_l = bk_i)\end{aligned}$$

where sk_I is the intruder open variable deemed to identify the secret shared with S_i. Checking KSL again with ψ_{KSL} and ϕ'_{KSL} yields some attacks. A brief analysis shows that they have in common an underlying assumption: The server shares the same key with I and A (or B), namely $sk_I = ak_i = sk_j$ ($sk_I = bk_i = sk_l$). When this happens, the intruder can easily open cryptograms generated by A (B), and steal the session key. Real servers are careful in choosing good keys, so those attacks may be regarded as false. Nevertheless, we want to modify ϕ'_{KSL} to forbid such connections. Again, the verification process can be tuned in order to rule out this case. This is done by further specifying the formula ϕ_{KSL}:

$$\begin{array}{rl}\phi''_{KSL} = \forall S.i: & (ak_i \neq bk_i) \wedge (a_i = \mathbf{I} \rightarrow ak_i = sk_I) \wedge (b_i = \mathbf{I} \rightarrow bk_i = sk_I) \wedge \\ & \exists A.j : (a_i = A_j \rightarrow sk_j = ak_i) \wedge \exists B.l : (b_i = B_l \rightarrow sk_l = bk_i)\end{array}$$

Repeating the search now yields no attacks.

6.1.5 Breaking KSL repeated authentication

In this section we show how a known attack can be found in the repeated authentication part of KSL and exemplify how the search for particular kinds of attacks can be performed. Following the outlined methodology we write the cIP template of principal A and B which represent step (6) to (8) of the informal specification.

$$\begin{array}{rcl} A & \triangleq & (b, sk, tk)[\\ & & out(nma, \{b, A, sk\}_{tk}). \\ & & in(?mb, \{nma\}_{sk}). \\ & & out(\{mb\}_{sk})]; \\ B & \triangleq & (sk, tk)[\\ & & in(?ma, \{B, ?u, sk\}_{tk}). \\ & & out(nmb, \{ma\}_{sk}). \\ & & in(\{nmb\}_{sk})]; \end{array}$$

The assumptions on the repeated authentication part are:

- Principals A and his partner (whose identity is stored in b) share a symmetric session key.
- A has received a ticket encrypted with a key (stored in its own variable tk) known only by b.
- The intruder knows the tickets issued during the key exchange part.

We want to check for authentication. Whenever A is connected to B and has the correct ticket, the nonces nma and nmb must be sent and received without modifications. In $\mathcal{PL}$ this amount to saying:

$$\begin{array}{rl}\psi_{REP} = \forall B.j : \exists A.i : & (b_i = B_j \wedge u_j = A_i \wedge tk_i = tk_j) \quad \rightarrow \\ & (ma_j = nma_i \wedge mb_i = nmb_j)\end{array}$$

More precisely, the formula states that: Every instance of B involved in a protocol run with an instance of A ($b_i = B_j$ and $u_j = A_i$) shares with A a ticket ($tk_i = tk_j$). If this happen the nonce received by B (stored in ma_j) is the same sent by A (nma_i) in that session. The same must hold for the nonce sent by B.

The third step of the methodology is the definition of the connection formula. Here we formalise the first two assumptions of the protocol, indeed we are interested in those traces where A and B share a session key and a ticket. Therefore the intended invariant property is:

$$\begin{array}{rl}\phi_{REP} = \forall A.i : \forall B.j : & b_i = B_j \wedge tk_i = tk_j \quad \rightarrow \\ & sk_i = sk_j\end{array}$$

Every time A is connected with B and has a ticket, then A and B share the same session key stored in sk_i and sk_j. We can restrict the search only to those traces containing at least an instance of A and B by adding two more conjunct to the formula: $\exists B.m : true \wedge \exists A.l : true$.

The repeated authentication part is executed after the first part of the protocol. The intruder has already seen the ticket issued by instances of B and all messages previously exchanged. The task of the fourth methodology step is to define the intruder knowledge. What the intruder knows is:

I_0, I_0+, I_0-	messages always known.
$\{B_2, A_3, sk_{B_2}\}_{tk_{B_2}}$	ticket issued by B_2 for A_3.
$\{B_1, A_3, sk_{B_1}\}_{tk_{B_1}}$	ticket issued by B_1 for A_3.

This is indeed the content of the `ksl.kb` file. As we can see it is not an initial knowledge but a template for it. Indeed it contains variables which will be instantiated by the join operation to the correct values. Name of principal are fixed because they originated from a previous run of the protocol, namely the key exchange part.

Running $\mathcal{A}$SPASyA with the previous cIP template, $\mathcal{PL}$ properties and initial knowledge leads to sixteen attack traces. An extract of $\mathcal{A}$SPASyA output is reported in table 6.1, whereas a snapshot of the actual output can be found in Chapter 9.

Violated Constraints:	**Open Variables**
$(b_{A_3} = B_2)$ $(u_{B_2} = A_3)$ $(tk_{A_3} = tk_{B_2})$ $(mb_{A_3} \neq nmb_2)$	$sk_{A_3} = sk_{B_1} = sk_{B_2} = ks$ $tk_{B_1} = kb1$ $tk_{B_2} = kb2$
Knowledge:	**Model:**
$\{nmb_1\}_{ks}, \{nmb_2\}_{ks},$ $nmb_1, \{nma_3\}_{ks},$ $nmb_2, \{B_2, A_3, ks\}_{kb2},$ $nma_3, A_3, A_3^+, B_2, B_2^+, B_1, B_1^+,$ $\{B_2, A_3, ks\}_{kb2}, \{B_1, A_3, ks\}_{kb1},$ $I_0, I_0^+, I_0^-.$	$ma_{B_1}(\kappa) \rightarrow nmb_2$ $ma_{B_2}(\kappa) \rightarrow nma_3$ $mb_{A_3}(\kappa) \rightarrow nmb_1$
Intruder:	
(1) $A_3 \rightarrow I : nma_3, \{B_2, A_3, ks\}_{kb2}$ (2) $I \rightarrow B_2 : nma_3, \{B_2, A_3, ks\}_{kb2}$ (3) $B_2 \rightarrow I : nmb_2, \{nma_3\}_{ks}$ (4) $I \rightarrow B_1 : nmb_2, \{B_1, A_3, ks\}_{kb1}$ (5) $B_1 \rightarrow I : nmb_1, \{nmb_2\}_{ks}$ (6) $I \rightarrow B_2 : \{nmb_2\}_{ks}$ (7) $I \rightarrow A_3 : nmb_1, \{nma_3\}_{ks}$ (8) $A_3 \rightarrow I : \{nmb_1\}_{ks}$ (9) $I \rightarrow B_1 : \{nmb_1\}_{ks}$	

Table 6.1: Attack report for KSL repeated authentication part.

The attack is performed exploiting the presence of two tickets issued for A by two different instances of B containing the same session key. This is highlighted in the connection conditions for the context in Table 6.1 ($sk_{A_3} = sk_{B_1} = sk_{B_2} = ks$). The intruder is able to authenticate itself as A_3 with B_1 and in doing so let A_3 and B_2 perform a bad sequence of message exchanges. In steps (1) to (3) A_3 and B_2 begin the authentication phase which is possible because of the tickets in the knowledge instatiated with the correct values. In steps (4) and (5), I uses B_1 as an oracle to encrypt nonce nmb_2 with ks. With that information I can match the input data requested by B_2. The attacker subsequently uses A_3 as an encrypting oracle to obtain $\{nmb_1\}_{ks}$ which is needed to end the protocol run with B_1. Hence the intruder has been able to let B_1 believe he was A_3.

Some other attacks found by $\mathcal{A}$SPASyA are permutations of the explained attack. There are some reported attacks that are possible under different connections. One of them is possible when there have been two sessions of KSL and the issued tickets not only contain the same session key but are also encrypted with the same key. Indeed, the connection formula specified does not negate this possibility even if it is quite unusual.

Chapter 7

$\mathcal{A}$SPASyA at Work

This section is a library of examples about protocol certification with $\mathcal{A}$SPASyA. Every protocol is described informally and then a cIP formalisation is given, together with security properties. Performance measurement[1] and reported attacks are also presented. For every protocol we report the verification time, the state space size (number of configurations) and average branching factor under different initial connections and increasing number of instances. We introduce some syntactic sugar to express connection formulae: We denote with σ a connection formula that holds in contexts that contain at least an instance of each role of the protocol. Consider a protocol with three roles, A,B and S. Then, $\sigma^{(A,B,S)}$ stands for $(\exists B.x : true) \wedge (\exists A.y : true) \wedge (\exists S.z : true)$. We also introduce ρ which is a connection formula that restrict the search to a single initial context. For example we will write $\rho^{(2A,1B,2S)}$ to denote the connection formula that holds only in a context with two instances of A, one instance of B and two instances of S.

7.1 Needham-Schroeder and KSL protocols

In this section we report the analysis results of the Needham-Schroeder and KSL protocols introduced in Chapter 6. They have been the chosen test protocols throughout the developing phase.

Needham Schroeder protocol performances are reported in the following table:

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
2	$true$	1	158	1.66	yes
3	$true$	3	5.183	1.75	yes
4	$true$	4:49	374.905	1.83	yes
4	$\sigma^{(A,B)}$	2:15	92.813	1.57	yes
4	$\rho^{(2A,2B)}$	12	18.385	1.52	yes
5	$true$	16:50:40	50.195.179	1.86	yes
6	$\rho^{(3A,3B)}$	23:52:48	57.896.853	1.57	yes

As we can see, the benefits of the constrained join are evident. Indeed, verifying the protocol with 4 instances yields very different computation times depending on the connections used: A full search requires two times the effort spent with $\sigma^{(A,B)}$ which in turn is definitely slower than $\rho^{(2A,2B)}$.

The following table reports the results of the tests performed on the KSL key exchange part with connection formulae presented in Chapter 6:

[1]Tests have been performed on an Athlon 2400+ with 1 Gb of ram.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
3	$\phi_{KSL} \wedge \rho^{(1A,1B,1S)}$	1	184	1.12	no
3	$\phi'_{KSL} \wedge \rho^{(1A,1B,1S)}$	10	2.775	1.32	yes
3	$\phi''_{KSL} \wedge \rho^{(1A,1B,1S)}$	1	373	1.13	no
3	$\phi''_{KSL} \wedge \sigma^{(A,B,S)}$	3:38:24	1829	1.15	no
5	$\phi''_{KSL} \wedge \rho^{(2A,1B,1S)}$	58	4.021	1.16	no
5	$\phi''_{KSL} \wedge \rho^{(1A,2B,1S)}$	1:52	27.326	1.36	no

It is worth to notice that using ϕ_{KSL}, the connection formula that does not take into consideration the intruder as a possible partner of the server S, the state space is very small. On the contrary, when more hypothesis are stated with ϕ'_{KSL} a bigger state space is generated containing traces generated by the omitted intruder capability. On the contrary, ϕ''_{KSL} is more restrictive and generates less states.

We also report tests performed on the KSL repeated authentication part:

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
2	$true$	1	279	1.1	no
2	$\phi_{REP} \wedge \sigma^{(A,B)}$	1	119	1.1	no
3	$\phi_{REP} \wedge \sigma^{(A,B)}$	3	5.584	1.13	yes
4	$\phi_{REP} \wedge \rho^{(2A,2B)}$	3:51	155.220	1.22	no

7.2 ISO symmetric key two-pass unilateral authentication protocol

This protocol belongs to a family of authentication mechanisms presented in [1]. It aims at unilateral authentication of a principal A to a principal B. The informal specification is:

$$\begin{array}{lll} (1) & B \rightarrow A: & Rb, Text1 \\ (2) & A \rightarrow B: & Text3, \{Rb, B, Text2\}_{k^{ab}} \end{array}$$

It is assumed that A and B share the symmetric key k^{ab}. B sends a random number (*Rb*) to A together with a text field; in the original presentation advices are given on the use of text fields, but for the sake of the verification it is not important to examine the details. A sends back a cryptogram containing the random number and B identity, hence principal B may assume that A is communicating.

The protocol presents some aspects that have to be carefully formalised:

- cIP semantics does not allow messages to be numbers. Random numbers can be represented as nonces, since they are used to ensure freshness and uniqueness of a message.
- Text fields are pieces of data that differ in every session of the protocol. They can also be represented as nonces for the purposes of this verification.

The corresponding cIP template is then:

$$\begin{array}{lll} A & \triangleq & (b, sk)[in(?r, ?td).out(na, \{r, b, nc\}_{sk})] \\ B & \triangleq & (sk)[out(nb, nd).in(?ta, \{nb, B, ?tc\}_{sk})] \end{array}$$

where nb plays the role of *Rb* and nonces nd, nc, na represent text fields.

The intended security property is that every time A wants to authenticate itself to B, datum *Rb* (nb in cIP) originated from B is correctly received by A and viceversa ($Text2$ which is nc in cIP).

$$\psi = \quad \forall A.i : \exists B.j : \\ b_i = B_j \rightarrow r_i = nb_j \wedge tc_j = nc_i$$

Indeed, the security formula ψ states that every instance of A willing to communicate with an instance of B ($b_i = B_j$) must receive the random number generated by B ($r_i = nb_j$), and B must receive the correct nonce ($tc_j = nc_i$).

Open variables have to be joined to let A and B share a symmetric key. In $\mathcal{PL}$:

$$\phi = \forall A.i : \exists B.j : b_i = B_j \rightarrow sk_i = sk_j$$

Indeed, ϕ amounts to say that every two communicating instances of A and B have their open variables instantiated with the same symmetric key ($sk_i = sk_j$).

Running $\mathcal{A}$SPASyA with the above specifications and an empty knowledge yields an attack. The attack found is performed in a context with two instances of A (A_1, A_2) that share the same key with B_3. The intruder obviously can swap the message originating from instances of A. Therefore B_3 is not able to discriminate. Indeed, ψ is false when $b_1 = B_3$ but $tc_3 \neq nc_1$.

In the following table we report verification results:

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
2	$\sigma^{(A,B)} \wedge \phi$	1	17	1.42	no
3	$\sigma^{(A,B)} \wedge \phi$	1	132	1.21	yes
4	$\rho^{(2A,2B)} \wedge \phi$	1	291	1.24	yes
6	$\rho^{(3A,3B)} \wedge \phi$	3:51	33.580	1.41	yes

7.3 Yahalom protocol

The Yahalom protocol (presented in [11]) has been designed to achieve authentication involving a trusted third party. The informal specification follows, together with the corresponding cIP template:

$$\begin{array}{lll} (1) & A \rightarrow B : & A, na \\ (2) & B \rightarrow S : & B, \{A, na, nb\}_{k^{bs}} \\ (3) & S \rightarrow A : & \{B, k^{ab}, na, nb\}_{k^{as}}, \{A, k^{ab}\}_{k^{bs}} \\ (4) & A \rightarrow B : & \{A, k^{ab}\}_{k^{bs}}, \{nb\}_{k^{ab}} \end{array}$$

$$\begin{array}{lcl} A & \triangleq & (b, sk)[out(A, na).in(\{b, ?r, na, ?tb\}_{sk}, ?c).out(c, \{tb\}_r)] \\ B & \triangleq & (sk)[in(?a, ?ta).out(B, \{a, ta, nb\}_{sk}).in(\{a, ?r\}_{sk}, \{nb\}_r)] \\ S & \triangleq & (u, uk, v, vk)[in(u, \{v, ?tv, ?tu\}_{uk}).out(\{u, kab, tv, tu\}_{vk}, \{v, kab\}_{uk})] \end{array}$$

In message (1) A starts the protocol by sending B its own identity and a nonce na. B asks S to generate a new session key k^{ab} sending a cryptogram (in message (2)) encrypted with k^{bs}, a symmetric key shared with S. The trusted server sends A a cryptogram encrypted with a shared key, containing the session key k^{ab} and the nonces na, nb to ensure freshness. The second cryptogram of message (3) is then forwarded to B to exchange k^{ab}. A shows to be alive by sending back nb that was generated by B in the current session, encrypted with the session key.

The authentication property involves, as usual, relations among exchanged data:

$$\begin{array}{ll} \psi = & \forall A.j : \exists B.l : \exists S.i : \\ & (u_i = B_l \wedge v_i = A_j \wedge b_j = B_l) \rightarrow \\ & (r_j = r_l \wedge r_l = kab_i \wedge ta_l = tv_i \wedge \\ & tv_i = na_j \wedge tu_i = tb_j \wedge tb_j = nb_l) \end{array}$$

The antecedent of the implication states the connections of A and B to the server ($u_i = B_l \wedge v_i = A_j$) and the willingness of A to communicate with B ($b_j = B_l$). With this premise, exchanged data (session key and nonces) must be properly communicated like kab_i that must be passed to A_j and B_l ($r_j = r_l = kab_i$).

As for KSL (Section 6.1.3), the invariant property states the correct connection between the server S and the principals A and B:

$$\begin{aligned}\phi = \; & \forall S.i : \\ & (\exists B.j : u_i = B_j \rightarrow uk_i = sk_j) \wedge \\ & (\exists A.l : v_i = A_l \rightarrow vk_i = sk_l) \wedge \\ & (uk_i \neq vk_i)\end{aligned}$$

As we can see from the following table no attacks are found. Indeed this version of the protocol suffers only (according to the author knowledge) of a subtle type flaw attack that is based on the use of structured keys, and therefore can not be captured with the formal framework.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
3	$\sigma^{(A,B,S)} \wedge \phi$	2:20	2.640	1.37	no
3	ϕ	2:11:56	15.980	1.33	no
4	$\sigma^{(A,B,S)} \wedge \phi$	1:46:54	58.264	1.36	no

7.4 Carlsen protocol

This protocol presented in [12] falls in the same category of the Yahalom protocol. We give the informal specification:

$$\begin{array}{lll} (1) & A \rightarrow B: & A, na \\ (2) & B \rightarrow S: & A, na, B, nb \\ (3) & S \rightarrow B: & \{k^{ab}, nb, A\}_{k^{bs}}, \{na, B, k^{ab}\}_{k^{as}} \\ (4) & B \rightarrow A: & \{na, B, k^{ab}\}_{k^{as}}, \{na\}_{k^{ab}}, nc \\ (5) & A \rightarrow B: & \{nc\}_{k^{ab}} \end{array}$$

As usual, it is assumed that S shares symmetric key k^{as} with A and k^{bs} with B. This protocol has some interesting design improvements w.r.t the Yahalom protocol:

- Symmetrically breaking messages. Indeed the message in (3) consists of two cryptogram where data is distributed into different patterns. This way, many type flaw attacks can be avoided, as discussed in [12].

- Minimal number of cryptograms. Symmetric keys shared with the server are used every time there is a request for authentication. This gives a real attacker, the possibility to mount a cryptoanalysis attack once that enough cryptograms are collected. This protocol uses only two such cryptograms.

The resulting cIP template is:

$$\begin{array}{lll} A & \triangleq & (b, sk)[out(A, na).in(\{na, b, ?r\}_{sk}, \{na\}_r, ?tb).out(tbr)] \\ B & \triangleq & (sk)[in(?a, ?ta).out(a, ta, B, nb).in(\{?r, nb, a\}_{sk}, ?c).out(c, \{ta\}_r, nc).in(\{nb\}_r)] \\ S & \triangleq & (u, uk, v, vk)[in(u, ?tu, v, ?tv).out(\{kab, tv, u\}_{vk}, \{tu, v, kab\}_{uk})] \end{array}$$

The security property is:

$$\begin{aligned}\psi = \quad & \forall A.i : \exists B.j : \exists S.l : \\ & (u_l = A_i \wedge v_l = B_j \wedge b_i = B_j) \rightarrow \\ & (a_j = A_i \wedge r_i = r_j \wedge r_j = kab_l \wedge \\ & tb_i = nb_j \wedge ta_j = na_i \wedge \\ & tu_l = na_i \wedge tv_l = nb_j)\end{aligned}$$

The formula contains many terms but is similar to the Yahalom one. Indeed, the premise is the same, while the consequent of the implication again states the correctness of messages exchange. For instance, it states that nonce na_i originating from an instance of A must be received by an instance of B ($ta_j = na_i$) and then forwarded to a S ($tu_l = na_i$).

The correct connection is the same as the Yahalom one. Indeed the protocols share the same assumptions.

$$\begin{aligned}\phi = \quad & \forall S.l : \\ & (\exists A.i : u_l = A_i \rightarrow sk_i = uk_l) \wedge \\ & (\exists B.j : v_l = B_j \rightarrow sk_j = vk_l) \wedge \\ & (uk_l \neq vk_l)\end{aligned}$$

The Carlsen protocol do not have known attacks and $\mathcal{A}$SPASyA have been used to test its correctness with at most 4 principal instances.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
3	$\sigma^{(A,B,S)} \wedge \phi$	13	534	1.18	no
3	$true$	1:02	9.792	1.19	no
4	$\sigma^{(A,B,S)} \wedge \phi$	2:29	8.138	1.18	no

7.5 Needham-Schroeder signature protocol

This protocol has been presented in [11]. It is used to let a principal A send a message to B guaranteeing its origin and integrity. A trusted third party is used. The informal specification is:

$$\begin{array}{lll} (1) & A \rightarrow S : & A, \{cs\}_{k^{as}} \\ (2) & S \rightarrow A : & \{A, cs\}_{k^{ss}} \\ (3) & A \rightarrow B : & message, \{A, cs\}_{k^{ss}} \\ (4) & B \rightarrow S : & B, \{A, cs\}_{k^{ss}} \\ (5) & S \rightarrow B : & \{A, cs\}_{k^{bs}} \end{array}$$

A calculates cs, a digest value of $message$ (usually with a one-way function) and sends it to S. The server then releases an authenticator i.e. a cryptogram encrypted with a key known only to the server itself hat contains the message digest. A can now send the message together with the authenticator. When B wants to check for integrity, simply asks S to decrypt the authenticator (4). S sends back (5) the message digest encrypted by means of a key shared with B. B can now calculate the digest of the message received and check against the digest sent by S.

The formal framework does not allow the use of one-way functions, but message digests can be represented with nonces as an approximation. Indeed we want to check that every time a datum is created by A, it is received by B through S in the same session without modifications. We can not express the dependency between cs and the original $message$. Abstracting away from the real nature of cs, needs a careful approach: In the following we will show how to analyse the results of the verification to ensure that found attacks depend on the protocol and not on the formalisation assumptions. Following these observations we obtain:

$$
\begin{array}{lcl}
A & \triangleq & (sk)[out(A, \{na\}_{sk}).in(?c).out(nm, c)] \\
B & \triangleq & (sk)[in(?m, ?c).out(B, c).in(\{?a, ?ta\}_{sk})] \\
S & \triangleq & (u, uk, v, vk)[in(u, \{?tu\}_{uk}).out(\{u, tu\}_{kss}).in(v, \{?a, ?ta\}_{kss}).out(\{a, ta\}_{vk})]
\end{array}
$$

where na plays the role of cs and nm stands for the message.

The security properties is:

$$
\begin{array}{rl}
\psi = & \forall A.i : \exists B.j : \exists S.l : \\
& u_l = A_i \wedge v_l = B_j \rightarrow \\
& u_l = a_l \wedge tu_l = ta_l \wedge a_j = a_l \wedge \\
& ta_j = ta_l \wedge tu_l = na_i
\end{array}
$$

It states the relation among data in the same session, without taking into consideration the dependency between nm and na but only the correct exchange of na that is received by a server communicating with $A(tu_l = na_i)$ and forwarded to B $(ta_j = ta_l)$. to

As the other protocols with a trusted third party already seen, the connection formula is:

$$
\begin{array}{rl}
\phi = & \forall S.l : \\
& (\exists A.i : u_l = A_i \rightarrow sk_i = uk_l) \wedge \\
& (\exists B.j : v_l = B_j \rightarrow sk_j = vk_l) \wedge \\
& (uk_l \neq vk_l)
\end{array}
$$

$\mathcal{A}$SPASyA finds an attack to this protocol whenever two instances of A (A_1, A_2) share the same key with S. Indeed the intruder can exchange digest messages and let B accept a message originated form A_1 even if it was created by A_2. We report a fragment of the attack where, for the sake of clarity, cIP nonces are substituted with their actual meaning:

$$
\begin{array}{lll}
(1) & A_1 \rightarrow I : & A_1, \{digest_1\}_{kas} \\
(2) & A_2 \rightarrow I : & A_2, \{digest_2\}_{kas} \\
(3) & I \rightarrow S_3 : & A_1, \{digest_2\}_{kas} \\
(4) & S \rightarrow I : & \{A_1, digest_2\}_{kss} \\
(5) & I \rightarrow A_1 : & \{A_1, digest_2\}_{kss} \\
(6) & I \rightarrow A_2 : & \{A_1, digest_2\}_{kss} \\
(7) & A_1 \rightarrow I : & message_1, \{A_1, digest_2\}_{kss} \\
(8) & A_2 \rightarrow I : & message_2, \{A_1, digest_2\}_{kss} \\
(9) & I \rightarrow B : & message_2, \{A_1, digest_2\}_{kss} \\
(10) & B \rightarrow I : & B, \{A_1, digest_2\}_{kss} \\
(11) & I \rightarrow S : & B, \{A_1, digest_2\}_{kss} \\
(12) & S \rightarrow I : & \{A_1, digest_2\}_{kbs} \\
(13) & I \rightarrow B : & \{A_1, digest_2\}_{kbs}
\end{array}
$$

As we can see, the intruder make B accept $message_2$ as if it were originated by A_1. This attack is not based on the dependence between messages and digests (which indeed is not violated) but on the lack of dependency between the digest and the identity of the initiator. This example shows how $\mathcal{A}$SPASyA can be effective even in the verification of an approximated protocol.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
3	$\sigma^{(A,B,S)} \wedge \phi$	10	860	1.2	no
3	$true$	1:08	4.350	1.17	no
4	$\rho^{(2A,1B,1S)} \wedge \psi$	16	13.507	1.2	yes

7.6 Denning-Sacco key distribution protocol

This protocol relies upon asymmetric cryptography to exchange a symmetric session key between two principals, using a trusted server. The informal specification given in [17] is:

$$\begin{array}{lll} (1) & A \rightarrow S: & A, B \\ (2) & S \rightarrow A: & Cert_A, Cert_B \\ (3) & A \rightarrow B: & Cert_A, Cert_B, \{\{k^{ab}\}_{A^-}\}_{B^+} \end{array}$$

where $Cert_X = \{X, X^+\}_{S^-}$ is a certificate provided by S. A certificate is encrypted with the private key of S, s hat anyone can decrypt it by means of the public key S^+, certifying in this way S as the generator of the cryptogram.

The cIP specification follows:

$$\begin{array}{lcl} S & \triangleq & ()[in(?x, ?y).out(\{x, x^+\}_{S^-}, \{y, y^+\}_{S^-})] \\ A & \triangleq & (x, y)[out(A, y).in(\{A, A^+\}_{x^+}, \{y, ?z\}_{x^+}).out(\{A, A^+\}_{x^-}, \{y, z\}_{x^-}, \{\{kab\}_{A^-}\}_z)] \\ B & \triangleq & (x)[in(\{?w, ?y\}_{x^+}, \{B, B^+\}_{x^+}, \{\{?r\}_y\}_{B^-})] \end{array}$$

The security property states the secrecy of the key kab, received by B by means of the r variable, and authentication of A to B:

$$\psi = \forall B.i : r_i \notin \mathfrak{K} \wedge (\forall B.j : \exists (A.k : y_j = A_k) \rightarrow (y_k = B_j))$$

Security formula ψ enforces that for every instance of B the received key must not be in the intruder knowledge and if the received certificate contains information about an instance of A ($y_j = A_k$) then that instance must be willing to communicate with B ($y_k = B_j$).

An obvious attack is possible when the intruder plays the role of S, but this is against the hypothesis of S being trusted. A connection formula is sufficient i order to avoid this case:

$$\phi = \forall A.j : \forall B.i : x_i \neq \mathbf{I} \wedge x_j \neq \mathbf{I} \wedge y_j \neq x_j$$

Indeed, we want to exclude from the search all the traces where A or B are communicating with I as trusted server ($x_i \neq \mathbf{I}, x_j \neq \mathbf{I}$) and where A is trying to authenticate with the trusted server ($x_j \neq y_j$).

$\mathcal{A}$SPASyA discovers an attack on this protocol:

Violated Constraints:	**Open Variables:**
$r_{B_1} \in \mathfrak{K}$	$x_{A_2} = S_4$
	$x_{B_1} = S_3$
	$y_{A_2} = S_3$
Knowledge:	**Model:**
$\{\{k_2\}_{A_2^-}\}_{S_3^+}, \{S_3, S_3^+\}_{S_4^-},$	$x_{S_3}(\kappa) \rightarrow S_4$
$\{A_2, A_2^+\}_{S_4^-}, \{S_4, S_4^+\}_{S_3^-},$	$y_{S_3}(\kappa) \rightarrow B_1$
$\{B_1, B_1^+\}_{S_3^-}, S_4, S_4^+, S_3, S_3^+, A_2,$	$y_{S_4}(\kappa) \rightarrow A_2$
$A_2^+, B_1, B_1^+, I_0, I_0^+, I_0$	$x_{S_4}(\kappa) \rightarrow S_3$
Intruder:	
$(1) A_2 \rightarrow I : A_2, S_3$	
$(2) I \rightarrow S_3 : S_4, B_1$	
$(3) S_3 \rightarrow I : \{S_4, S_4^+\}_{S_3^-}, \{B_1, B_1^+\}_{S_3^-}$	
$(4) I \rightarrow S_4 : S_3, A_2$	
$(5) S_4 \rightarrow I : \{S_3, S_3^+\}_{S_4^-}, \{A_2, A_2^+\}_{S_4^-}$	
$(6) I \rightarrow A_2 : \{A_2, A_2^+\}_{S_4^-}, \{S_3, S_3^+\}_{S_4^-}$	
$(7) A_2 \rightarrow I : \{A_2, A_2^+\}_{S_4^-}, \{S_3, S_3^+\}_{S_4^-}, \{\{k_2\}_{A_2^-}\}_{S_3^+}$	
$(8) I \rightarrow B_1 : \{S_4, S_4^+\}_{S_3^-}, \{B_1, B_1^+\}_{S_3^-}, \{\{S_3, S_3^+\}_{S_4^-}\}_{B_1^+}$	

The attack is found in a context with two server instances. The intruder playing the part of S_4, is able to share with B_1 a session key. This is possible because in step (3) the intruder manages to obtain certificates from S_3 issued for S_4 and B_1. Then, in step (5) the intruder also obtain certificates from S_4 issued for S_3 and A_2. Subsequently, exploiting the knowledge of the certificates, the intruder can authenticate himself as S_4 with B_1 using (S_3, S_3^+) as session key. Indeed, this can be classified as a type flaw attack with constructed key that can nonetheless be reported by $\mathcal{A}$SPASyA.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
3	$\sigma^{(A,B,S)} \wedge \phi$	2	6.416	3.3	no
4	$\rho^{(1A,1B,2S)} \wedge \phi$	1:21	127.323	4.0	yes
5	$\rho^{(2A,1B,2S)} \wedge \phi$	3:08:24	11.748.962	4.48	yes
5	$\rho^{(1A,2B,2S)} \wedge \phi$	36:05	2.415.759	4.89	yes
6	$\rho^{(2A,2B,2S)} \wedge \phi$	107:12:46	308.622.707	5.5	yes

Denning-Sacco protocol is the most computationally expensive protocol we tested: Its average branching factor is high mostly because certificates are involved. Indeed, the intruder can decrypt each sent certificate building different certificates with his own key and subsequently exploit them with the principals he is connected to. Moreover there are many open variables representing principal names and used to denote asymmetric keys that give rise to numerous transitions for a single action, thus generating the observed branching factor.

7.7 Beller-Yacobi protocol

This protocol has been designed in [3], to let a mobile base B exchange a session key with a control point A. The informal specification follows:

$$\begin{array}{lll} (1) & A \rightarrow B: & A, A^+ \\ (2) & B \rightarrow A: & \{k^{ab}\}_{A^+} \\ (3) & A \rightarrow B: & \{na\}_{k^{ab}} \\ (4) & B \rightarrow A: & \{B, B^+, Cert_B, \{na\}_{B^-}\}_{k^{ab}} \end{array}$$

When a mobile B enters a new cell it tries to authenticate with the control point. The control point A sends B its identity together with its public key. B generates the session key k^{ab} and sends the received public key to A. In step (4) B shows its own identity by sending back in response to (3) a cryptogram containing information on its own identity, public key, a certificate issued by a trusted third party, and the nonce na, received in (3), encrypted with its own private key. A problem with this protocol is evident even from the informal specification: There is nothing that ensures the identity of A in step (1), allowing everyone to play the role of A whose public key is known.

The cIP specification is:

$$\begin{array}{lll} A & \triangleq & (b,s)[out(A, A^+).in(\{?r\}_{A^-}).out(\{na\}_r).in(\{b, ?bk, \{b, bk\}_s, \{na\}_{b^-}\}_r)] \\ B & \triangleq & (s)[in(?a, ?ak).out(\{kab\}_{ak}).in(\{?ta\}_{kab}).out(\{B, B^+, \{B, B^+\}_s, \{ta\}_{B^-}\}_{kab})] \end{array}$$

The trusted third party identity is shared between A and B by means of the open variable s. Indeed s represents a private key known by the third party and used to sign the issued certificates. In general, the formal framework does not allow for asymmetric keys to be stored in open variables. We analyse the protocol representing s as a symmetric key and showing that the reported attacks are independent from the details of the certificate.

The desired security property requires the authentication of A to B and the secrecy of the session key:

$$\begin{aligned}\psi = \quad & \forall B.i : (kab_i \in \mathfrak{K} \rightarrow a_i = \mathbf{I}) \wedge \\ & (\exists A.j : b_j = B_i \rightarrow \\ & (r_j = kab_i \wedge bk_j = B_i^+ \wedge ta_i = na_j \wedge kab_i \notin \mathfrak{K}))\end{aligned}$$

In each session, the session key kab_i generated by B_i must remain secret unless the intruder is behaving as a honest principal connected to B_i. When B_i is communicating with an instance A_j, willing to authenticate to B_i ($b_j = B_i$), then nonce na_j and key kab_i must be communicated correctly.

A connection formula is needed to restrict the analysis to the cases in which A and B share the same third party certificate:

$$\begin{aligned}\phi = \quad & \forall A.i : (\exists B.j : b_i = B_j \rightarrow s_i = s_j) \vee \\ & (\exists A.l : b_i = A_l \rightarrow s_i = s_l) \vee (b_i = \mathbf{I})\end{aligned}$$

$\mathcal{A}$SPASyA finds several attacks, all originated from the design error spotted before. A representative intruder is:

Violated Constraints:	**Open Variables:**
$na_2 \in \mathfrak{K}, b_{A_2} \neq I_0,$	$b_{A_2} \rightarrow B_1$
$kab_1 \in \mathfrak{K}, a_{B_1} \neq I_0,$	$s_{A_2} \rightarrow ks$
$b_{A_2} = B_1, r_{A_2} \neq kab_1$	$s_{B_1} \rightarrow ks$
Knowledge:	**Model:**
$\{B_1, B_1^+\}_{ks}, kab_1, na_2,$	$r_{A_2}(\kappa) \rightarrow kI_0$
$A_2, A_2^+, B_1, B_1^+, I_0,$	$ak_{B_1}(\kappa) \rightarrow kI_0$
I_0^+, I_0^-, kI_0	$bk_{A_2}(\kappa) \rightarrow B_1^+$
Intruder:	
$(1) A_2 \rightarrow I : A_2, A_2^+$	
$(2) I \rightarrow A_2 : \{kI_0\}_{A_2^+}$	
$(3) A_2 \rightarrow I : \{na_2\}_{kI_0}$	
$(4) I \rightarrow B_1 : A_2, I^+$	
$(5) B_1 \rightarrow I : \{kab_1\}_{I^+}$	
$(6) I \rightarrow B_1 : \{na_2\}_{kab_1}$	
$(7) B_1 \rightarrow I : \{B_1, B_1^+, \{B_1, B_1^+\}_{ks}, \{na_2\}_{B_1^-}\}_{kab_1}$	
$(8) I \rightarrow A_2 : \{B_!, B_1^+, \{B_1, B_1^+\}_{ks}, \{na_2\}_{B_1^+}\}_{kI_0}$	

It is important to note that $\mathcal{A}$SPASyA finds all the possible attacks by instantiating variables in all possible ways. For instance, an attack where kI_0 is substituted with I^+ or I^- is reported, together with an attack where nonce na is replaced by an intruder generated nonce.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
2	ϕ	1	848	1.54	yes
3	ϕ	1:55	139.359	1.48	yes
4	$\phi \wedge \rho^{(2A,2B)}$	9:22:36	6.326.775	1.61	yes

Recently in [6] using stating analysis techniques, an improved version of this protocol, called the Beller-Yacobi MSR protocol, has been showed to present an unknown new flaw. In the following we report the specification of the protocol and we show that $\mathcal{A}$SPASyA is able to find the same attack.

The informal specification of the MSR protocol is:

$$
\begin{array}{lll}
(1) & A \rightarrow B: & A, \{A\}_{S^-}, A^+ \\
(2) & B \rightarrow A: & \{k^{ab}\}_{A^+} \\
(3) & B \rightarrow A: & \{B, BS^-\}_{k^{ab}}
\end{array}
$$

Message (1) has been modified by adding a certificate issued by a trusted third party S, containing the certification of the identity. As before, B generates a session key k^{ab} and sends it back to A using its public key received in step (1). In message (3) B sends the certificate of its own identity encrypted with the session key.

The cIP specification is:

$$
\begin{array}{lll}
A & \triangleq & (b,s)[out(A,\{A\}_s, A^+).in(\{?r\}_{A^-}).in(\{b,\{b\}_s\}_r)] \\
B & \triangleq & (s)[in(?a,\{a\}_s, ?ak).out(\{kab\}_{ak}).out(\{B,\{B\}_s\}_{kab})]
\end{array}
$$

The desired security property is represented by a slightly modified ψ where the references to nonces are removed:

$$
\begin{array}{ll}
\psi' = & \forall B.i : (kab_i \in \mathfrak{K} \rightarrow a_i = \mathbf{I}) \wedge \\
& (\exists A.j : b_j = B_i \rightarrow \\
& (r_j = kab_i \wedge a_i = A_j \wedge kab_i \notin \mathfrak{K}))
\end{array}
$$

Open variables and connections are unchanged and hence ϕ is used again as the connection formula.

Performing a search with $\mathcal{A}$SPASyA yields some attacks. Among these we can find the aforementioned flaw as presented in the following attack report.

Violated Constraints:	**Open Variables:**
$b_{A_2} = B_1$,	$b_{A_2} \rightarrow B_1$
$a_{B_1} \neq A_2$	$s_{A_2} \rightarrow s_{B_1} \rightarrow ks$
Knowledge:	**Model:**
$\{B1,\{B1\}_{ks}\}_{kab_1}, \{kab_1\}_{A_2^+}, \{A_2\}_{ks}, A_2, A_2^+,$	$a_{B_1}(\kappa) \rightarrow A_3$
$B_1, B_1^+, \{A_3\}_{ks}, A_3, I_0, I_0^+, I_0^-$	$ak_{B_1}(\kappa) \rightarrow A_2^+$
Intruder:	
$(1)A_2 \rightarrow I : A_2, \{A_2\}_{ks}, A_2^+$	
$(2)I \rightarrow B_1 : A_3, \{A_3\}_{ks}, A_2^+$	
$(3)B_1 \rightarrow I : \{kab_1\}_{A_2^+}$	
$(4)B_1 \rightarrow I : \{B_1, \{B_1\}_{ks}\}_{kab_1}$	
$(5)I \rightarrow A_2 : \{kab_1\}_{A_2^+}$	
$(6)I \rightarrow A_2 : \{B_1, \{B_1\}_{ks}\}_k ab_1$	

As we can see from the report, the intruder is able to let B_1 authenticate himself to A_2 believing to be communicating with A_3. This is possible because the certificate does not contain any reference to the public key of the principal whose identity is certified.

7.8 Bilateral key exchange protocol

This protocol has been designed to let two principals A and B exchange a symmetric key using public key cryptography. The informal specification given in [14] is:

$$
\begin{array}{lll}
(1) & B \rightarrow A: & B, \{nb, B\}_{A^+} \\
(2) & A \rightarrow B: & \{nb, na, A, k\}_{B^+} \\
(3) & B \rightarrow A: & \{na\}_k
\end{array}
$$

The corresponding cIP template is:

$$\begin{array}{lcl} A & \triangleq & ()[in(?b, \{?tb, b\}_{A^-}).out(\{tb, na, A, kab\}_{b^+}).in(\{na\}_r)] \\ B & \triangleq & (a)[out(B, \{nb, B\}_{a^+}).in(\{nb, ?ta, a, ?r\}_{B^-}).out(\{ta\}_r)] \end{array}$$

The desired security property is:

$$\begin{array}{rl} \psi = & \forall B.i : (\exists A.j : a_i = A_j \rightarrow \\ & tb_j = nb_i \wedge ta_i = na_j \wedge r_i = kab_j \\ & b_j = B_i \wedge r_i \notin \mathfrak{K}) \vee (a_i = \mathbf{I}) \end{array}$$

The corresponding search yields no attack. Indeed, there are no attacks on this protocol (as far as author knowledge is concerned) and $\mathcal{A}$SPASyA verify correctness up to 4 principal instances.

Instances	Connection	Time (hh:mm:ss)	Conf.	B.F.	Attack
2	*true*	1	191	2.53	no
3	*true*	5	6.316	3.1	no
4	*true*	7:48	373.587	3.7	no
4	$\rho^{(2A,2B)}$	44	65.793	3.3	no

Chapter 8

$\mathcal{A}$SPASyA and Friends

In this chapter we review some verification techniques and tools for the verification of security protocols and we compare $\mathcal{A}$SPASyA with them. Many different approaches have been presented throughout the literature, like theorem proving, finite state machines, and static analysis (see Chapter 1). A reasonable comparison can be done by focusing our attention to the tools that implement a symbolic model checking approach. Among these, the most important (in the author knowledge) are STA [8], TRUST [43, 42] and PCS [31]. However, our analysis will be more focused on methodological comparisons than on efficiency or search strategies. This kind of approach is justified by the fact that the compared tools implement different semantics and different logics, making the measurements on the size of the generated state space less important. Indeed, each tool has a different concept of state; thus, states of different tools may contain different information and may require different amount of computation to be generated. Moreover, properties must be verified and the verification cost (in terms of efficiency and number of verifications) is greatly affected by the state. We think that it is relevant to have a tool that allows the user to clearly specify the assumptions of the protocols and to easily spot errors in the formalisation phase.

Some remarks and comparisons between $\mathcal{A}$SPASyA and techniques different from model checking are reported in Section 8.3

8.1 Symbolic model checkers

We briefly introduce STA, TRUST, and PCS describing their features and highlighting merits and drawbacks (in our opinion) of their related methodologies.

8.1.1 STA: Symbolic Trace Analyser

STA, described in [8], uses symbolic techniques for trace generation and relies upon a dialect of the spi-calculus [2] to represent principal behaviour. The calculus provides primitives to represent symmetric and asymmetric encryption with atomic keys, and hashing. Properties are specified with a logic of *correspondence*. More precisely STA allows properties of the form $\alpha \hookleftarrow \beta$ where α and β are principal actions, meaning that for every generated trace, if β occurs in the trace, then α has occurred at some previous point in the same trace, namely β corresponds to α.

STA has a very efficient implementation and offers the possibility to specify the initial knowledge of the intruder. However STA lacks the possibility of template definition. This means that the user has to specify by hand every principal instance giving rise to a long and error prone protocol definition. The absence of templates has an impact on certification effectiveness. Indeed, many protocol assumptions depend on the initial intruder knowledge and principal connections, and without templates the user has to specify them by hand. Even for simple protocols this can lead to not consider some interesting scenarios. For instance, referring to Section 6.1.5, the repeated authentication part of KSL has a flaw when different tickets contain the same session key. In STA one should state this hypothesis by adding to the knowledge the incriminated tickets specifying by hand their content, whereas in $\mathcal{A}$SPASyA a template for a class of ticket is given and, during the join process, every hypothesis is automatically generated and checked, preventing omissions. This drawback, is particularly evident in principal connections. For instance, the STA specification presented in [8] of the Needham Schroeder protocol defines the process of the initiator A as a non-deterministic sum of a process willing to communicate with B and a process willing to communicate with the intruder. With this specification STA cannot automatically find the type flaw attack presented in Section 6.1.2, as $\mathcal{A}$SPASyA does, forcing the user to give a different formalisation for each different connection of principals. Finally, the logic of correspondence is powerful enough to express interesting properties of protocols like secrecy and authentication, but we believe it may be awkward when used to express more complex properties.

8.1.2 TRUST

TRUST has been presented in [42] and subsequently enhanced in [43]. Principals are expressed with a name-passing calculus in the style of π-calculus. Principals have a sort of open variables that are instatiated with principal names, and can use primitives for encrypting data with symmetric and asymmetric atomic keys. Properties are embedded in principal definitions as a set of assertion to be verified during state space exploration.

The implemented reduction semantics is very efficient and optimised, leading to fast verification of protocol sessions with relatively many instances. TRUST presents the same problems of STA concerning templates for initial knowledge, because they are specified without any reference to principal open variables. Regarding principal connections, TRUST is definitely better than STA because, using open variables and assertions, one can select interesting initial contexts even if there is not a clear mechanism to make principals share secrets. We believe that the main drawback in TRUST is property specification. Without a precise separation of concerns, protocol formalisation can be misleading. Indeed, assertion are embedded in principal definitions and a change in the property to be checked may lead to a complete re-formalisation of principal behaviour. One interesting aspect of TRUST is an optimisation to the reduction semantics that, recording information on messages exchanged along a trace, can detect when two traces give rise to the same interaction pattern and are therefore equivalent. With this method, TRUST analyses only one trace for each "class of equivalence" greatly reducing the state space. Moreover, this approach leads to smaller reports for attacks allowing for fast analysis of results.

8.1.3 PCS

PCS [31] stands for Prolog Constraint Solver and is a tool based on the *strand space* model presented in [39, 40, 13]. A strand is a sequence of actions that represents the behaviour of either a principal or an intruder. A strand space is a set of strands together with a graph structure given by a causal dependence. Properties are expressed in terms of connections between strands of different kinds. A strand can be parameterised with variables and a trace is generated finding a substitution for which an interaction graph exists. PCS is not a proper model checker, but it follows a very similar approach. Indeed, principals and hence strands, are represented with terms of a free algebra that embeds the notion of symmetric and asymmetric cryptography with structured keys and hashing. Properties are specified by adding to the strand space, a strand representing what the intruder must not be able to do. For instance, in the Needham Schroeder protocol nonces na and nb must remain secret.

To check this property one adds to the strand space an intruder that tries to acquire *nb* or *na*. If a suitable substitution is found by the constraint solving procedure, which satisfies the requirements of each strand (including the intruder's one), the intruder can steal the nonces and an attacks is found.

In PCS there are devices very similar to the join mechanism of $\mathcal{A}$SPASyA but there is not the possibility for the user to impose constraints on principal connections. Initial knowledge specification is given by adding data to the strand space, and can be fully parameterised with variables. We believe that the main drawback of PCS is the specification of properties that, as in TRUST, are not clearly separated from principal behaviour.

8.2 Summing up

In this section we aim at showing that we further developed verification methodologies trying to correct unsatisfactory aspects of other approaches. Indeed, $\mathcal{A}$SPASyA provides:

- Clear separation of protocol specification and properties to be tested, which otherwise may give rise to an error prone protocol formalisation.
- Means to formalise the implicit assumptions present in the informal specification of a protocol, which are crucial for the correctness of verification.
- Intuitive way to specify intruder's power, which is a significant means to analyse a protocol under weaker conditions.
- Template-based protocol specification to avoid omission of interesting scenarios and hypotheses.
- Mechanisms for selective expansion of the state space by invariant pruning.

On the other hand, $\mathcal{A}$SPASyA has some weak points that we hope to cope with in future work. Indeed, as the number of instances grows, the join mechanism, deemed to generate initial hypotheses on open variables, needs a large amount of computational time and become a bottleneck compared with state expansion and formula verification. Another issue is the size of the reported results, which can be very large and therefore not easily checkable, due to the big number of permutations of the same attack trace. The formal framework also need some straightforward extensions to encapsulate more useful primitives as hashing, non atomic keys and time-stamps.

Despite these limitations, we think that $\mathcal{A}$SPASyA is an advance with respect to STA, TRUST and PCS mostly because it allows for a precise representation of principal behaviour totally separated from the property to be verified. Also, intruder power and knowledge can be precisely specified as well as principal connections. Moreover, we think that $\mathcal{PL}$ logic is well suited to express relations among principal data and the principal that exchange them, and can be clearer than correspondence assertions, especially when mixed with principal specification. Table 8.1 collects the previously discussed merits and drawbacks of the analysed tools and $\mathcal{A}$SPASyA.

Regarding efficiency issues, $\mathcal{A}$SPASyA is quite efficient and can handle a number of instances comparable to other tools. The performance measurements are quite difficult to compare with other tools. Indeed, they have different semantics leading to different state spaces. For comparison purposes we reported in Table 8.2 verification times for TRUST and $\mathcal{A}$SPASyA. We can notice, confronting with [8, 42], that STA and TRUST have the fastest reduction engines, however $\mathcal{A}$SPASyA considers a bigger state space involving more initial contexts and therefore checking more scenarios.

Features	STA	TRUST	PCS	$\mathcal{A}$SPASyA
Templates for principals	$\times$	$\surd$	$\surd$	$\surd$
Templates for knowledge	$\times$	$\times$	$\surd$	$\surd$
Principal connections	$\times$	$\times$	$\times$	$\surd$
Embedded properties	$\surd$	$\surd$	$\surd$	$\times$
Non atomic key cryptography	$\times$	$\times$	$\surd$	$\times$
Hashing	$\surd$	$\surd$	$\surd$	$\times$
Number of sessions for most protocols	3	3	$-$	3

Table 8.1: Synthetic comparisons of tools. $\surd$ and $\times$ stand for provided and not provided features, whereas $-$ is an unknown datum. Data have been gathered from [42, 43, 31, 8].

Protocol	TRUST	$\mathcal{A}$SPASyA
Needham Schroeder (3 instances)	0.50	3
Needham Schroeder (4 instances)	22	2:15
Yahalom (3 instances)	12	2:20

Table 8.2: Verification time comparisons between TRUST [42] and $\mathcal{A}$SPASyA.

8.3 Other tools

For the sake of completeness we briefly review some tools based on different approaches. We focus on aspects that are related with $\mathcal{A}$SPASyA in terms of methodology.

8.3.1 Finite state model checkers

Before the introduction of symbolic techniques, model checking approaches were limited to the verification of finite state machines. In finite state model checking, principal behaviour is formalised by means of a formal calculus and the transition rules are restricted to generate a finite state space. Hence, the number of principal instances is bounded as well as the complexity of messages. Indeed, limiting the number of encryption used to build a message yields to a finite state space. The finite state model checker FDR, has been used by Lowe [28] to discover an unknown flaw in the Needham Schroeder protocol. Another important tool reported in the literature is murϕ [18]. Symbolic analysis is a further development of this approach.

8.3.2 A theorem prover

Many theorem provers have been adapted and modified to verify security protocols. Recently Blanchet [4] has developed a tool based on theorem proving techniques. Principal behaviour is specified with a formal calculus in the style of π-calculus. A compiler is then used to translate the specification into a set of logic clauses that are fed the theorem prover. The tool provides hashing and many different cryptographic primitives. As for PCS, principals and knowledge are parameterised with variables but the mechanism for the verification is different. Indeed, on some classes of security protocols Blanchet's tool is non terminating, due to the nature of processing. However, it certifies protocol with an unbounded number of principals (provided that the termination is reached).

8.3.3 A static analyser

As pointed out in Chapter 1, static analysis techniques have been successfully applied to protocol certification. We refer to a tool based on LySA [5], a calculus similar to π-calculus that supports a unique global channel for communication between processes. Principals are represented as LySA processes with annotations, namely conditions that must hold at certain control points. The representation is automatically translated in a suitable logic and subsequently solved with theorem proving techniques. The tool reports a super-set of values that variables can assume at control points, and no explicit representation of the intruder behaviour is given. Moreover, there might be false reports of attacks just because static analysis approximates the solutions. However, this approach has been successfully applied [6] finding a new attack on the Beller-Yacobi [3] protocol. As shown in Section 7.7, $\mathcal{A}$SPASyA has been able to find the same flaw and to report the intruder process.

Chapter 9

User Manual

This chapter describes how to run $\mathcal{A}$SPASyA. We describe command line parameters and file formats for input and output.

9.1 Using $\mathcal{A}$SPASyA

To certify a protocol, the user must specify in the following order:

- A cIP description of roles, contained in a text file, with a **.pr** extension,
- a security formula in $\mathcal{PL}$ format contained in a **.pl** file,
- a connection formula in $\mathcal{PL}$ format contained in a **.pj** file,
- the list of messages belonging to the initial intruder knowledge, in a **.kb** file.

To start a verification session, the user also must provide the number m of principal instances specified by the `-nprinc` m switch. Moreover verification mode must be enabled by means of the `-V` switch. The general syntax of the command is:

```
aspasya file.pr file.pl file.pj file.kb -V -nprinc m switches
```

The switches have the following semantics:

- `-i` enables interactive search. Every time an attack is found, the user can choose to stop searching, continue searching in the same trace (more than one attack is possible) or in the next trace.
- `-f` enables search feedback. Useful information is printed during the search. The user can be aware of the estimated verification time, the number of attack found, the current sub-tree of the state space tree and the program status (searching or verifying). Moreover $\mathcal{A}$SPASyA prints out a table with performance data (computation time, configurations generated on termination, average branching factor).
- `-mga` enables automatic multiple attack skip. If this option is specified, $\mathcal{A}$SPASyA searches the state space and reports only one attack per trace.
- `-s` *`parameters`* allows the specification of the initial context. Even if one could select only one initial context for expansion by means of a complex connection formula, this switch allows one to directly specify the number of instances for each role. The n^{th} argument of `-s` is the number of instances of the n^{th} role specified in the `.pr` file. Refer to Section 9.3 for example of use.

- `-g` selects the symbolic name expansion during join operation. Indeed the standard join algorithm assigns to each open variable v representing a name the corresponding symbolic variable $v^P(\kappa)$. With this option, every name in κ is directly assigned to v. This generates more initial contexts (that may eventually be pruned), but allows for a small branching factor during trace expansion.
- `-H` selects `html` mode. Attacks are reported in a html page, more readable than the ASCII report.

It is possible to simply print out the state space without verifying the protocol. This is specified with `-T` or `-G` options. The former prints out every single trace in a text file while the latter produces a file in GraphViz [1] format.

9.2 Specifying templates

$\mathcal{A}$SPASyA is equipped with parsers to let the user specify protocol data without worrying about internal representation. Generally speaking, the user have to specify cIP templates for principal representation, invariant and security properties written in $\mathcal{PL}$ and messages belonging to the initial intruder knowledge.

cIP principals and $\mathcal{PL}$ formulae are specified i th input files `.pr`, `.pl` and `.pj`, with a syntax very similar to the one defined in the formal framework and reported in tables 9.1 and 9.2.

PRINCIPAL	::=	NAME: (VARIABLES) [PROCESS];
NAME	::=	any uppercase strings.
VARIABLES	::=	a comma separated list of VARIABLE
VARIABLE	::=	any lowercase string not beginning with 'k' or 'n'
PROCESS	::=	ACTION ||
		PROCESS.PROCESS ||
		PROCESS + PROCESS ||
		PROCESS | PROCESS
ACTION	::=	in(MESSAGE) || out(MESSAGE)
MESSAGE	::=	(MESSAGE, MESSAGE) ||
		{MESSAGE}MESSAGE ||
		KEY || NAME || NAME+ ||
		NAME- || NONCE ||
		VARIABLE || ?VARIABLE ||
		VARIABLE+ || VARIABLE-
KEY	::=	any lowercase string beginning with 'k'
NONCE	::=	any lowercase string beginning with 'n'

Table 9.1: cIP grammar.

Every datum in a $\mathcal{PL}$ formula is indexed with a index name that will be substituted to the correct instance value during formula normalisation. Some constants may be indexed with the correct instance value preceded by the corresponding principal name, as in MESSAGE_NAME_NUMBER of Table 9.2. Relations among data are expressed with `=` (equality) and `:>` (derivability), and are composed with quantifiers and a rich set of boolean operators to build $\mathcal{PL}$ formulae. The content of `.kb` file is a list of I-MESSAGE separated by commas. The initial knowledge cannot contain messages with literal indexes, but only messages indexed with instance names and numbers. Refer to Section 9.3 for an extended example.

In every input file, comments are allowed as in standard C^{++} language: A single line comment is denoted by `//` whereas multiline comment begins with `/*` and ends with `*/`.

[1] Refer to `http://www.research.att.com/sw/tools/graphviz/` for more information.

FORMULA	::=	I-VARIABLE = I-MESSAGE ||
		I-VARIABLE <> I-MESSAGE ||
		:> I-MESSAGE ||
		!:> I-MESSAGE ||
		`forall` NAME.INDEX: FORMULA ||
		`exists` NAME.INDEX: FORMULA ||
		! FORMULA ||
		FORMULA & FORMULA ||
		FORMULA | FORMULA ||
		FORMULA => FORMULA ||
		(FORMULA) ||
		`true` || `false`
INDEX	::=	a lowercase letter
I-VARIABLE	::=	VARIABLE_INDEX || VARIABLE_NAME_NUMBER
I-MESSAGE	::=	MESSAGE_INDEX || MESSAGE_NAME_NUMBER

Table 9.2: $\mathcal{PL}$ grammar.

9.3 A verification session

This section exemplifies step by step a verification session for the KSL repeated authentication part as introduced in Section 6.1.5.

The first step is the definition of the `kslrep.pr` file:

```
//KSL repeated authentication part.
A: (b,sk,tk) [
      out((nma,{((b,A),sk)}tk)).
      in((?mb,{nma}sk)).
      out({mb}sk)];

B: (sk,tk) [
      in((?ma,{((B,?u),sk)}tk)).
      out((nmb,{ma}sk)).
      in({nmb}sk)];
```

Security and invariant formulae are specified in `kslrep.pl` and `kslrep.pj` files:

```
//kslrep.pl
forall B.j:
   exists A.i:
     b_i = B_j and u_j = A_i  and tk_i = tk_j =>
       ma_j = nma_i and
       mb_i = nmb_j;


//kslrep.pj
(exists A.o: true) and
(exists B.o: true) and
forall A.i:
   forall B.j:
     b_i = B_j and tk_i = tk_j =>
       sk_i = sk_j;
```

Finally, the initial knowledge containing issued tickets is given in `kslrep.kb`

```
//kslrep.kb
I_0,                         /* intruder name */
{((B_2,A_3), sk_B_2)}tk_B_2,  /* ticket from B2 to A3 */
{((B_1,A_3), sk_B_1)}tk_B_1;  /* ticket from B1 to A3 */
```

Once the protocol input data is ready, we can run $\mathcal{A}$SPASyA to certify KSL. For instance we can give the following commands "`aspasya kslrep.pr kslrep.pl kslrep.pj kslrep.kb`" with the following options:

`-nprinc 3 -V` starts a search with 3 principal instances and outputs results on the standard output in `ascii` format;

`-nprinc 3 -V -f -i` starts a search with 3 instances giving remaining time estimate during the search and printing a table of useful data on the standard error. Moreover, `-i` enables interactive mode, and the user will be prompted for continuation after finding an attack;

`-nprinc 5 -V -mga -s 3 2 -H` starts a search with 5 instances, reporting in html only the first attack for every attack trace. With `-s 3 2` the search will be limited to the context made of 3 instances of A and 2 instances of B;

`-nprinc 3 -T` will produce the description of every trace in ascii format.

Whenever an attack is found, $\mathcal{A}$SPASyA produces information about the incriminated traces. An attack report is structured in several points:

Violated constraints. A list of the false conjuncts of the security property is displayed. This is a peculiarity of a model checker which is always able to give counterexamples.

Configuration. A snapshot of the incriminated trace is reported by displaying the knowledge at the end of the trace, the cIP processes of principals instantiated with the the global substitution χ and the assignment entailed by formula satisfaction.

Intruder reconstruction. The representation of intruder behaviour is displayed in the informal notation.

In Figure 9.1, we report a screen-shot of $\mathcal{A}$SPASyA verifying the KSL repeated authentication part together with the search data reported by the `-f` option. Finally, Figure 9.2 is a screen-shot of the corresponding attack report generated by the `-H` option.

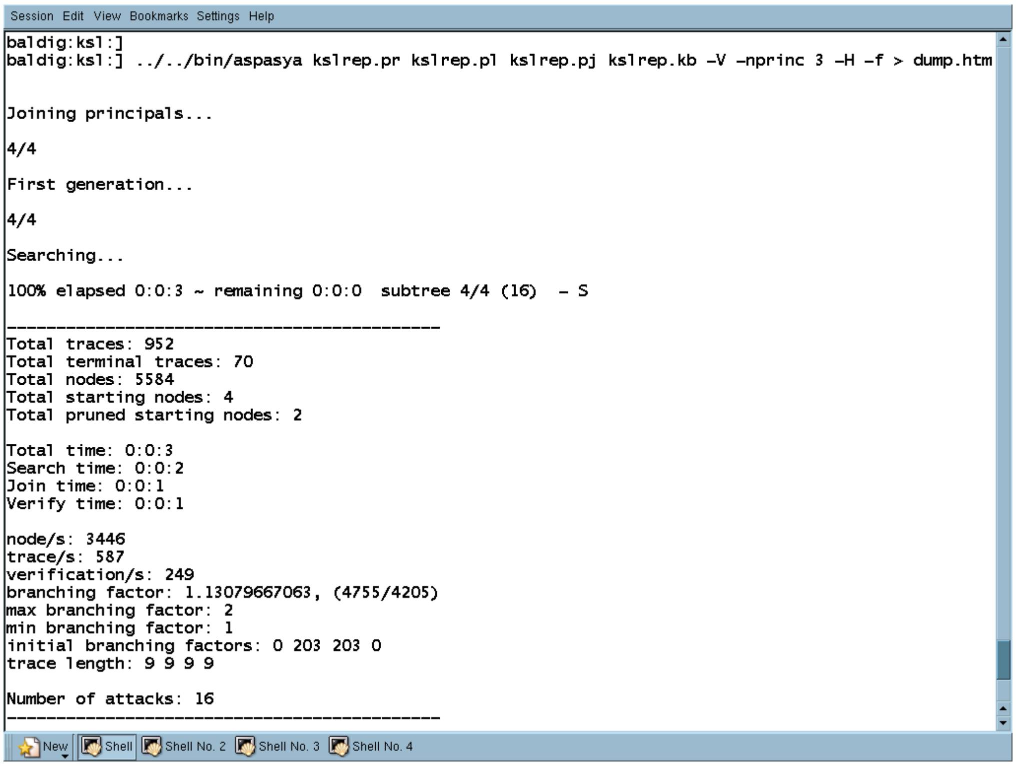

Figure 9.1: $\mathcal{A}$SPASyA at work with options "`-nprinc 3 -V -f -H`" on the KSL repeated authentication part.

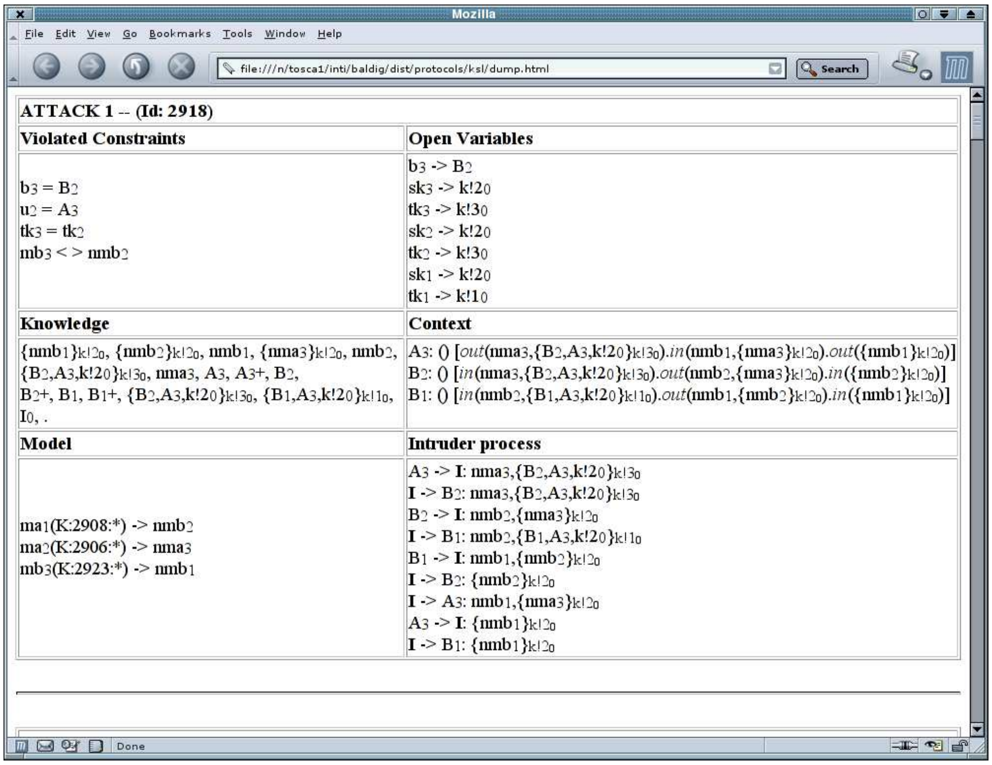

Figure 9.2: The attack report in html.

Chapter 10

Conclusions

In this thesis we have introduced $\mathcal{A}$SPASyA, a tool for protocol verification based on a symbolic model checking approach together with a methodology for the verification of security protocols; the methodology has been applied for analysing many protocols.

The development of $\mathcal{A}$SPASyA has been inspired by the theoretical framework presented in [10, 41], where an interesting modelisation of agent behaviour in open systems is given. The main ingredients of the theory are:

- The cIP formal calculus. Each cIP process represents a protocol principal, where cryptographic primitives are embedded in communication actions, and *open variables* allow for a parameterised connection between principals. Principals can be connected to others by the *join* operation. It embeds cryptographic primitives in principal actions.
- The $\mathcal{PL}$ logic used to state the desired properties of protocol. It allows us to express relations between the exchanged data, the principals and their roles in the protocol, the way in which they share secrets and the intruder knowledge.
- A symbolic semantics that models the evolution of principal communications in a hostile environment, exploiting symbolic techniques to make the verification procedure effective.

The separation of concerns of the theoretical framework has been adopted in the implementation as well. We developed $\mathcal{A}$SPASyA implementing each aspect of the verification framework as a separated module and then realise the verification algorithm by letting all modules interact. From the implementation point of view we achieved two main goals:

- The modular architecture can be easily expanded both by adding new functionalities and by applying the environment to other fields.
- An efficient model checker with performances comparable to the similar existent tools.

The most important feature of our work is the verification methodology related to $\mathcal{A}$SPASyA. Indeed, an effective verification tool should focus on limiting as much as possible the sources of errors in the formalisation process. Hence, we developed an incremental methodology that allows for the iterative refinement of protocol specification. It consists of four steps in which the user specifies:

1. The representation of principals,
2. the desired security property,

3. the connections among principals and
4. the initial intruder knowledge.

Each step is clearly separated from the others and involves different aspects of the formalisation. For instance, the principal behaviour is given in the first step and it is never changed in the others. Moreover, the third and fourth steps of our methodology allows for the tuning the search a nd the power of the intruder in an intuitive way. Indeed, principal connections can be constrained by means of a $\mathcal{PL}$ formula and the intruder power can be augmented by adding information to its initial knowledge. The latter device is particularly useful to discover attacks where the intruder exploits information about previous sessions of the protocol. The constrained join mechanisms has been introduced during the development phase and constitutes an enhancement of the theoretical framework. Concluding, from the verification methodology point of view, we reached the following goals, most of which are distinguishing features of our approach:

- Clear separation of protocol specification from the property to be tested, which otherwise may give rise to protocol mis-interpretation.
- Means of formalisation for implicit assumptions present in the informal specification of a protocol, which are crucial in correctness of verification.
- Intuitive way of specifying intruder's power, which is a powerful formal tool to analyse a protocol under different assumptions.
- Template based protocol specification to avoid omission of interesting scenarios and hypothesis.
- Mechanisms for selective expansion of the state space by invariant pruning.

Experimentally, we have applied the methodology to the verification of several protocols. We have found many known attacks on the investigated protocols, sometimes without being aware of some of the which were not so popular in the literature; we also have reported a very recent attack found by means of static analysis techniques in [6], on the Beller-Yacobi protocol [3] (Section 7.7). It is not clear if it also can be found with STA or TRUST. For most protocols we have spotted some design errors that can potentially lead to flaws (Section 7.5) using $\mathcal{A}$SPASyA to test the robustness of the protocols.

10.1 Future work

The functionalities of $\mathcal{A}$SPASyA can be enriched in several ways. First, the modularity of the architecture accounts for an easily integration of $\mathcal{A}$SPASyAas a component of the Profundis Web[1], an online service that gathers together and integrates verification tools based on process calculi. There are two further lines of work. On the one hand, we would like to extend $\mathcal{A}$SPASyA and its framework to handle non atomic keys, hashing functions and time-stamps. Perhaps this extension will require additional pruning strategies to further shrink the state space following an approach similar to [43]. On the other hand, $\mathcal{A}$SPASyA is based on a theory that models agents in open systems. The theory is naturally oriented to the modelisation of recently emerged problems, most notably the verification of *Web Services*. Web services are software applications that use the markup language XML to exchange data with other applications by using Internet protocols. Web services operate over any network to achieve specific tasks, called methods or functions, that other applications can invoke and use. This is a new vision for using the Internet in the development, engineering and use of software, aimed at revolutionizing distributed computing. Security properties must be guaranteed not only on the behaviour of web services but also on their connections. Hence, we believe that $\mathcal{A}$SPASyA can be adapted to be on of the pioneer tools in this field.

[1] Refer to `http://jordie.di.unipi.it:8080/pweb/index_html` for more information.

Bibliography

[1] ISO/IEC 9798. Information technology - security techniques - entity authentication mechanism part 2: Entity authentication using symmetrictechniques. Technical report, 1993.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[3] M.J. Beller and Y. Yacobi. Authentication and key agreement protocol for pcs. Technical report, P&A JEM/93012, November 1993.

[4] Bruno Blanchet. Automatic Verification of Cryptographic Protocols: A Logic Programming Approach (invited talk). In *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 1–3, Uppsala, Sweden, August 2003. ACM.

[5] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 126–140, 2003.

[6] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis can find new flaws too. In *Workshop on Issues on the Theory of Security (WITS'04), Barcelona, 2004, ENTCS, Elsevier*, 2004.

[7] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127:27–34, 2001.

[8] Michele Boreale. Symbolic trace analysis of cryptographic protocols. In *28th Colloquium on Automata, Languages and Programming (ICALP)*, LNCS. Springer, July 2001.

[9] Michele Boreale and Marzia Buscemi. A framework for the analysis of security protocols. In *CONCUR: 13th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2002.

[10] Andrea Bracciali. *Behavioural Patterns and Software Composition*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2003.

[11] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[12] Ulf Carlsen. Optimal privacy and authentication on a portable communications sytem. *Operating Systems Review*, 28(3):16–23, 1994.

[13] Cervesato, Durgin, Mitchell, Lincoln, and Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[14] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0. Available at `http://www.cs.fsu.edu/ yasinsac/group/work/childs/`, November 1997.

[15] Somesh Clarke, Edmund M. Jha and Wilfredo R. Marrero. Using state space exploration and a nautural deduction style message derivation engine to verify security protocols. In *In Proc.* IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998*, 1998.

[16] H. Comon and V. Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not, 2001.

[17] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(7):533–536, August 1981.

[18] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[19] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[20] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEETIT: IEEE Transactions on Information Theory*, 29, 1983.

[21] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol — version 3.0. Available at `http://home.netscape.com/eng/ssl3/ssl-toc.html`, March 1996.

[22] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, 2004.

[23] Dan Harkins and Dave Carrel. RFC 2409: The Internet Key Exchange (IKE), November 1998. Status: PROPOSED STANDARD.

[24] Antti Huima. Efficient finite-state analysis of security protocols. In *Formal methods and security protocols*, FLOC Workshop, Trento, 1999. INRIA.

[25] Axel Kehne, Jürgen Schönwälder, and Horst Langendörfer. Multiple authentications with a nonce-based protocol using generalized timestamps. In *Proc. ICCC '92*, Genua, 1992.

[26] J. Kohl and B. Neuman. The kerberos network authentication service (version 5). Internet Request for Comment RFC-1510, 1993.

[27] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, November 1995.

[28] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[29] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[30] Alfred J. Menzies, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[31] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[33] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[34] Clifford Neumann and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14, 1993.

[35] Lawrence C. Paulson. *The inductive approach to verifying cryptographic protocols*. Technical report; no. 443. 4006797499. University of Cambridge Computer Laboratory, Cambridge, UK, USA, February 1998.

[36] Zunino Roberto and Degano Pierpaolo. A note on the perfect encryption assumption in a process calculus. In *Foundations of Software Science and Computation Structures: 7th International Conference, FOSSACS 2004*, 2004.

[37] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessios is np-complete. RR 4134, Inria, 2001.

[38] Douglas R. Stinson. *Cryptography: Theory and practice*. CRC Press, 1995.

[39] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, 1998.

[40] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct, 1999.

[41] Emilio Tuosto. *Non-Functional Aspects of Wide Area Network Programming*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Via Buonarroti 2, 56125 Pisa - Italy, May 2003.

[42] Vincent Vanackére. The trust protocol analyser. automatic and efficient verification of cryptographic protocols. In *VERIFY02*, 2002.

[43] Vincent Vanackére. History-dependent scheduling for cryptographic processes, 2004.

[44] Tatu Ylonen. SSH — secure login connections over the Internet. In USENIX Association, editor, *6th USENIX Security Symposium, July 22–25, 1996. San Jose, CA*, pages 37–42, Berkeley, CA, USA, July 1996. USENIX.

Appendix A

Appendix: Source code

We report the source code of each module of $\mathcal{A}$SPASyA. In Figure A.1 the graph of dependencies between modules is also reported. The modules are :

Action contains the definition of labels for the transition system.

Assignment contains the definition of variable substitutions.

Context contains the definition of a context of principal instances.

Core contains the definition of the symbolic match procedures.

Csolver contains the definition of the constraint store and functions to solve it.

Gviz contains functions for Graphviz output.

Join contains functions to add new principal instances to running contexts.

Khash contains the definition of a hash table to store knowledge references.

Kmananger contains functions to correctly update the hash table of knowledges.

Knowledge contains the definition of the intruder knowledge.

Logic contains the definition of $\mathcal{PL}$ formulae and functions for their normalisation.

Message contains message definitions and methods to handle them.

Node contains the definition of a node of the state space tree

Parsecmdline contains the definition of the command line options.

Parser contains the definition of the cIP parser.

Pl_parser contains the definition of the $\mathcal{PL}$ formulae parser.

Principal contains the definition of cIP principals.

Princpool contains functions to correctly instantiate principal indexes after the join.

Process contains the definition of cIP processes.

Status contains the definition of configurations.

Step contains functions to make one step of computation along a trace.

Thestack contains the definition of a general purpose stack used in the search process.

Tokenizer contains the definition of a simple lexical analyser.

Tracer contains functions to generate and check the entire state space using `step`, `join` and `verifier`.

Utils contains useful functions on lists.

Verifier contains functions to check satisfability of formulae against configurations.

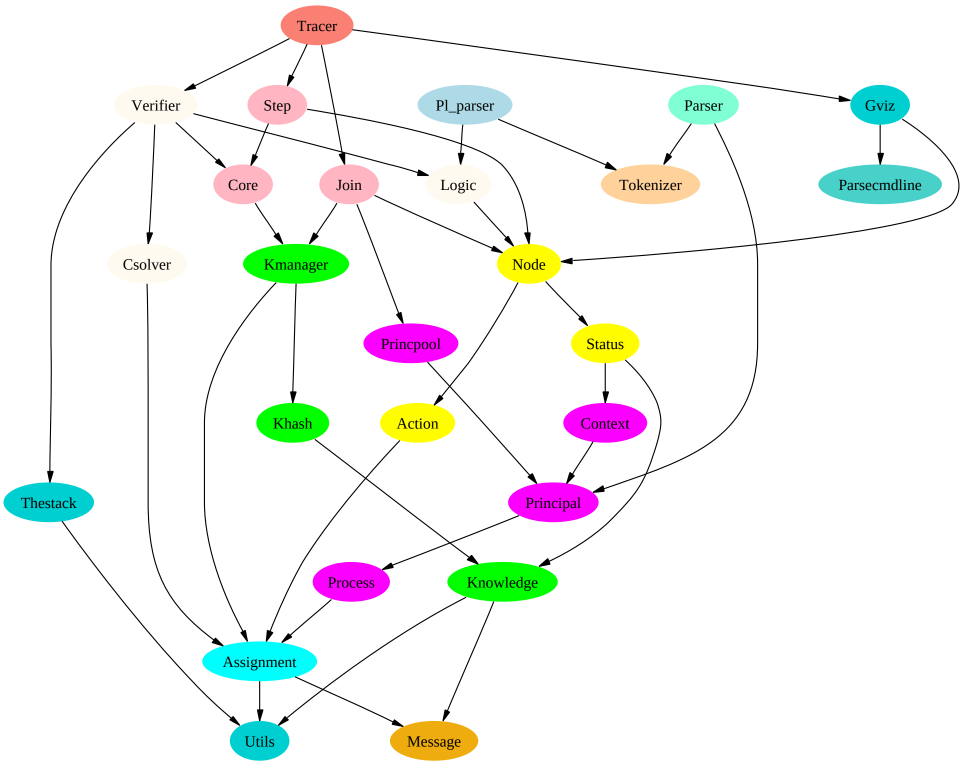

Figure A.1: Dependency graph of modules.

Action

```
open Message open Assignment

type action =
  | Input of pname*msg*assignment | Output of pname*msg*assignment | Join
  of pname*assignment | NoMoreAct | Closed

let equal =
  function x ->
    function y ->
      match x,y with | Input(p1,m1,s1), Input(p2,m2,s2) -> (s1 m1) = (s2 m2)
      | Output(p1,m1,s1), Output(p2,m2,s2) -> (s1 m1) = (s2 m2) | NoMoreAct,
      NoMoreAct -> true | _ -> false

let prnt =
```

```
  function a ->
    match a with | Input((sn,i),m,s) ->
        (
         print_string(sn^"_"^string_of_int(i)^": In("); prntMsg m;
         print_string(")  --sigma-->  In("); prntMsg (s m); print_string(")\n")
        )
    | Output((sn,i),m,s) ->
        (
         print_string(sn^"_"^string_of_int(i)^": Out("); prntMsg
         m; print_string(")  --sigma-->  Out("); prntMsg (s m);
         print_string(")\n")
        )
    | Join(pn,asg) ->
        (
         print_string("Join: "^fst(pn)^"_"^string_of_int(snd(pn))^"\n")
        )
    | NoMoreAct -> print_string("0\n")

let tostring =
        function a ->
                match a with
                        | Input((sn,i),m,s) -> sn^"_"^string_of_int(i)^":
                        I("^(Message.tostring (s m))^")" |
                        Output((sn,i),m,s) -> sn^"_"^string_of_int(i)^":
                        O("^(Message.tostring (s m))^")" | Join(p,s) ->
                        "J("^fst(p)^","^string_of_int(snd(p))^")" | NoMoreAct
                        -> "0"
```

Aspasya

```
open Message;; open Sys;; open Utils;; open Parsecmdline;; open Gc;;
open Tracer;;

let opts = init_options();; let continue=ref(true);;

if not (!Sys.interactive) then
  (
   if Array.length(Sys.argv)<2 then (synopsis false; continue:=false) else (
     (parsecmdline (Sys.argv) opts); if (opts.errmsg<>"") then
     (print_string("\n"^opts.errmsg^"\n"); continue:=false) else
       if opts.help then (synopsis true; continue:=false) else
         if opts.pr_file="" then (synopsis false; continue:=false) else
         (if opts.pl_file="" then (synopsis false; continue:=false) else
         (if opts.kb_file="" then (synopsis false; continue:=false) else ()))
    )
  )
else (synopsis false; continue:=false);;



let rec readall =
```

```
  function ct ->
    if (Parser.more_parsing()) then
      let p = Parser.parsePrincipal() in if (Parser.getErrorCode()<>0)
      then (print_string(Parser.getErrorMsg()); failwith "\nSyntax error in
      parsing principals\n") else ( let ct2 = p::ct in readall(ct2))
    else ct;;

if(!continue) then (

  Pl_parser.init_parser(opts.pl_file); let pl = Pl_parser.parseFormula() in if
  (Pl_parser.getErrorCode()<>0) then (print_string(Pl_parser.getErrorMsg());
  failwith "\nSyntax error in parsing formula\n") else ();
  Pl_parser.done_parser();


  Pl_parser.init_parser(opts.pj_file); let plj = Pl_parser.parseFormula() in if
  (Pl_parser.getErrorCode()<>0) then (print_string(Pl_parser.getErrorMsg());
  failwith "\nSyntax error in parsing join components\n") else ();

  Pl_parser.done_parser();

  Pl_parser.init_parser(opts.kb_file); let kbl = Pl_parser.parseKb() in if
  (Pl_parser.getErrorCode()<>0) then (print_string(Pl_parser.getErrorMsg());
  failwith "\nSyntax error in parsing knowledge\n") else ();
  Pl_parser.done_parser();

  Parser.init_parser(opts.pr_file); let prns = readall([]) in
  Parser.done_parser();


  let ss = init_search prns kbl opts in do_joins ss pl plj;

  if (opts.output_type = 1) then Gviz.initgraph "aspasya" else (); if
  (opts.output_type = 2 && opts.attack_type = 2) then Parsecmdline.htmlinit()
  else (); do_search ss; if (opts.output_type = 1) then Gviz.donegraph()
  else (); if (opts.output_type = 2 && opts.attack_type = 2) then
  Parsecmdline.htmldone() else ();


 ) else ()
```

Assignment

```
open Message open Utils


exception NoMeaning exception NeedVariable
```

```
type assignment = msg -> msg


let getfromtable =
  function table ->
    function v ->
      let res = filter table (function (vv,mm) -> v=vv) in if res = []
      then v else snd(head res)

let addtable =
  function table ->
    function (v,m) ->
      let rec closure =
        function tt ->
          function (vv,mm) ->
            let nt = filter tt (function (vvv,mmm) -> mmm=vv) in let ad =
            map nt (function (vvv,mmm) -> (vvv,mm)) in ad@(debox(map ad
            (function (vvv,mmm) -> closure tt (vvv,mmm))))
      in if (v=m) then table else
        match v with | Var(_) | SVar(_) -> (
            let nm = getfromtable table v in if nm = v then (v,m)::table
            else table
           )
        | _ -> raise NeedVariable


let rec f =
  function tt ->
    function n ->
      let rec pass =
        function table ->
          function m ->
            match m with | Var(p) -> getfromtable table m | SVar(p,i,t) ->
            getfromtable table m | BVar(p) ->
                let nm =getfromtable table (Var(p)) in if nm = Var(p) then
                m else nm
            | Couple(m1,m2) -> Couple(pass table m1, pass table m2) |
            Crypt(m1,key) -> Crypt(pass table m1, pass table key) | K(_) |
            PN(_) | PNp(_) | PNm(_) | NO(_) -> m | Publ(Var(v)) ->
                let vv = getfromtable table (Var(v)) in ( match vv with |
                PN(s,p) -> PNp(s,p) | SVar(vr,i,t) -> Publ(SVar(vr,i,t)) |
                Var(v) -> m | _ ->  Publ(vv)
               )
            | Priv(Var(v)) ->
                let vv = getfromtable table (Var(v)) in ( match vv with |
                PN(s,p) -> PNm(s,p) | SVar(vr,i,t) -> Priv(SVar(vr,i,t)) |
                Var(v) -> m | _ ->  Priv(vv)
               )
            | Publ(SVar(v,i,t)) when (t=Gen || t=GenR)->
                let vv2 = getfromtable table (SVar(v,i,t)) in ( match vv2
                with | PN(s,p) -> PNp(s,p) | SVar(v,i,Name) -> Publ(vv2)
                | SVar(v,i,RName) -> Publ(vv2) | SVar(v,i,t) -> m | _ ->
                Publ(vv2)
               )
```

```
          | Priv(SVar(v,i,t)) when (t=Gen || t=GenR)->
              let vv2 = getfromtable table (SVar(v,i,t)) in ( match vv2
              with | PN(s,p) -> PNm(s,p) | SVar(v,i,Name) -> Priv(vv2)
              | SVar(v,i,RName) -> Priv(vv2) | SVar(v,i,t) -> m | _ ->
              Priv(vv2)
             )
          | Publ(SVar(v,i,Name)) ->
              let vv2 = getfromtable table (SVar(v,i,Name)) in ( match
              vv2 with | PN(s,p) -> PNp(s,p) | SVar(v,i,Name) -> m |
              SVar(v,i,RName) -> Publ(vv2) | _ -> Publ(vv2)
             )
          | Priv(SVar(v,i,Name)) ->
              let vv2 = getfromtable table (SVar(v,i,Name)) in ( match
              vv2 with | PN(s,p) -> PNm(s,p) | SVar(v,i,Name) -> m |
              SVar(v,i,RName) -> Priv(vv2) | _ -> Priv(vv2)
             )
          | Publ(SVar(v,i,RName)) ->
              let vv2 = getfromtable table (SVar(v,i,RName)) in ( match
              vv2 with | PN(s,p) -> PNp(s,p) | SVar(v,i,RName) -> m | _
              -> Publ(vv2)
             )
          | Priv(SVar(v,i,RName)) ->
              let vv2 = getfromtable table (SVar(v,i,RName)) in ( match
              vv2 with | PN(s,p) -> PNm(s,p) | SVar(v,i,RName) -> m | _
              -> Priv(vv2)
             )
          | Priv(_) | Publ(_) -> m | Bottom -> m | _ -> raise NoMeaning
      in match n with | Trick(_) -> Trick(tt) | _ ->
          let newm = pass tt n in if newm = n then newm else f tt newm



let void =
  function () ->
    f []

let newasg =
  function table ->
    let checktable = filter table (function (v,m) -> v<>m) in f checktable

let isVoid =
  function asg ->
    let Trick(tab) = asg (Trick([])) in tab = []

let add =
  function asg ->
    function var ->
      function newmsg ->
        let Trick(tab) = asg (Trick([])) in let newtable = addtable tab
        (var,newmsg) in f newtable


let prntHtml =
  function asg ->
```

```
    let rec prnttab =
      function l ->
        match l with | (v,n)::xs -> (prntMsgHtml ((v)); print_string("
        -&gt "); prntMsgHtml n; print_string("<br>")); prnttab xs | [] ->
        print_string(" .<br>")
    in let Trick(tab) = asg (Trick([])) in prnttab tab


let prnt =
  function asg ->
    let rec prnttab =
      function l ->
        match l with | (v,n)::xs -> (prntMsg ((v)); print_string(" -> ");
        prntMsg n; print_string("\n")); prnttab xs | [] -> print_string(" .\n")
    in let Trick(tab) = asg (Trick([])) in prnttab tab


let tostring =
  function asg ->
    let rec prnttab =
      function ll ->
        match ll with | [] -> ("\\n"); | (v,n)::xs -> (Message.tostring v)^"
        -> "^(Message.tostring n)^("\\n")^(prnttab xs)
    in let Trick(tab) = asg (Trick([])) in prnttab tab


let ( * ) =
  function alfa ->
    function beta ->
      function m ->
        let Trick(t1) = alfa (Trick([])) in let Trick(t2) = beta (Trick([]))
        in let nt = t1@t2 in newasg nt m

let comp = ( * )

let listfy =
  function asg ->
    let Trick(tab) = asg (Trick([])) in tab

let leftof =
  function asg ->
    function v ->
      let sss = listfy asg in map (filter sss (function (a,b) -> b=v)) (fst)
```

Context

```
open Principal open Utils

type context = principal list
```

```
let void =
  function () -> []

let add =
  function (ct,np) -> np::ct

let ( ++ ) =
  function ct ->
    function (mp,np) ->
      let rec sublist = function (ls,el,tls) ->
        match ls with | [] -> (tls,[]) | x::xs when (x=el) -> (tls,xs) |
        x::xs -> sublist(xs,el,(x::tls))
      in let (head, tail) = sublist(ct,mp,[]) in (np::head)@(tail)

let rec getprincipal =
  function (ct,k) ->
    match ct with | x::xs -> if k = 1 then x else getprincipal(xs,k-1)

let getlist =
  function ct -> ct

let setlist =
  function ct -> ct

let getPrev =
  function ct -> List.tl ct

let isEmpty =
  function ct -> forall ct (function x -> Principal.isConsumed x)

let hasEmptyPrincipal =
  function ct -> exists ct (function x -> Principal.isConsumed x)

let howmany =
  function ct -> len ct

let getindexedprinc =
  function ct ->
    function i ->
      function asg ->
        let [ttt] = filter ct (function (nn,ii,vlist,expr) -> ii=i) in
        Principal.substitute asg ttt

let getopenprinc =
  function ct ->
    function asg ->
      let ttt = filter ct (function (nn,ii,vlist,expr) -> vlist<>[]) in let
      mmm = map ttt (function p -> Principal.substitute asg p) in mmm

let getIndexesOf =
  function ct ->
    function s ->
      let ttt = filter ct (function (nn,ii,vlist,expr) -> nn=s) in map ttt
```

```
      (function (nn,ii,vlist,expr) -> ii)

let rec getOpenVariables =
  function ct ->
    match ct with | [] -> (0,[]) | pp::pps ->
        let vs = Principal.getVars pp in let (nn,vv) = getOpenVariables pps in
        (nn+ len(vs),vs@vv)


let getlentrace =
  function ct ->
    if ct = [] then 0 else fold ct (function a -> function b -> Principal.len
    a + b) 0


let applyall =
  function ct ->
    function asg ->
      map ct (Principal.substitute asg )

let rec prnt =
  function ct ->
    function asg ->
      match ct with | [] -> print_string("\n") | x::xs -> Principal.prnt x
      asg; print_string("\n"); prnt xs asg

let rec prntHtml =
  function ct ->
    function asg ->
      match ct with | [] -> print_string("<br>") | x::xs -> Principal.prntHtml
      x asg; print_string("<br>"); prntHtml xs asg

let rec tostring =
  function ct ->
    function asg ->
      match ct with | [] -> "\\n" | x::xs -> (Principal.tostring x
      asg)^"\\n"^(tostring xs asg)
```

Core

```
open Message open Knowledge open Kmanager open Assignment open Utils

exception Undefined


(* Symbolic match functions *) let rec l =
  function kman ->
    function asg ->
      function mm ->
        match mm with | K(_) | PNp(_,_) | PNm(_,_) | NO(_,_) | PN(_,_) ->
```

```
[Assignment.void()] | SVar(v,i,_) -> [Assignment.void()] | BVar(v) ->
[Assignment.void()]

| Publ(SVar(v,i,RName)) | Priv(SVar(v,i,RName)) -> [Assignment.void()]
| Publ(SVar(v,i,Name)) | Priv(SVar(v,i,Name)) ->
    let kr = if (Knowledge.hasRNames (Kmanager.getK kman i)) then
    [(SVar(v,i,RName))] else [] in let kk = (Knowledge.getNames
    (Kmanager.getK kman i))@kr in let res1 = map kk (function pn ->
    Assignment.add (Assignment.void())  (SVar(v,i,Name)) (pn)) in res1
| Publ(SVar(v,i,t)) when (t=Gen || t=GenR) ->
    let res1 = [Assignment.add (Assignment.void()) (SVar(v,i,t))
    (SVar(v,i,Name))] in let res2 = (map res1 (function sigma ->
    l kman (asg) (sigma mm))) in rebuild res1 res2 (function (s1)
    -> function (s2) -> (s2*s1))
| Priv(SVar(v,i,t)) when (t=Gen || t=GenR) ->
    let res1 = [Assignment.add (Assignment.void()) (SVar(v,i,t))
    (SVar(v,i,Name))] in let res2 = (map res1 (function sigma ->
    l kman (asg) (sigma mm))) in rebuild res1 res2 (function (s1)
    -> function (s2) -> (s2*s1))

| Couple(m1,m2) ->
    let res1 = (l kman asg m1) in let res2 = (map res1 (function sigma
    -> l kman (asg) (sigma m2))) in rebuild res1 res2 (function (s1)
    -> function (s2) -> (s2*s1))

| Crypt(m, Publ(SVar(v,k,Name))) | Crypt(m, Priv(SVar(v,k,Name))) ->
    let kr = if (Knowledge.hasRNames (Kmanager.getK kman k)) then
    [(SVar(v,k,RName))] else [] in let kk = (Knowledge.getNames
    (Kmanager.getK kman k))@kr in let res1 = map kk (function pn ->
    Assignment.add (Assignment.void())  (SVar(v,k,Name)) (pn)) in
    let res2 = (map res1 (function sigma -> l kman (asg) (sigma m)))
    in rebuild res1 res2 (function (s1) -> function (s2) -> (s2*s1))

| Crypt(m, Publ(SVar(v,k,RName))) | Crypt(m, Priv(SVar(v,k,RName))) ->
    l kman asg m

| Crypt(m, SVar(v,k,t)) when (Knowledge.hasKeys (Kmanager.getK kman
k)) && (t=Gen || t=GenR) ->
    let oldk = (Kmanager.getK kman k) in let ks = if
    (Knowledge.hasSKeys oldk) then [SVar(v,k,SKey)] else [] in let
    kpm = if (Knowledge.hasPMKeys oldk) then [SVar(v,k,PMKey)] else
    [] in let kmp = if (kpm<>[]) then [SVar(v,k,MPKey)] else [] in
    let kps = Knowledge.getPKeys oldk in let kms = Knowledge.getMKeys
    oldk in let kk = ks@kpm@kmp@kps@kms in let res1 = map kk (function
    pn -> Assignment.add (Assignment.void())  (SVar(v,k,t)) (pn)) in
    let res2 = (map res1 (function sigma -> l kman (asg) (sigma m)))
    in rebuild res1 res2 (function (s1) -> function (s2) -> (s2*s1))

| Crypt(m, Priv(SVar(v,k,t))) | Crypt(m, Publ(SVar(v,k,t))) when
(t=Gen || t=GenR) ->
    let sr = Assignment.add (Assignment.void()) (SVar(v,k,t))
    (SVar(v,k,Name)) in let res = l kman asg (sr mm) in map res
    (function sigma -> sigma*sr)
```

```
        | Crypt(m, SVar(v,k,t)) -> l kman asg m | Crypt(m, kk) when
        isGroundKey kk -> l kman asg m | Crypt(m,Publ(_)) -> l kman asg m |
        Crypt(m,Priv(_)) -> l kman asg m | Crypt(m, _) -> l kman asg m | _ ->
        []
and mu =
  function kman ->
    function k ->
      function asg ->
        function mm -> (
          match mm with | BVar(v) when not (Knowledge.isEmpty k)
          -> [(Var(v), Assignment.add (Assignment.void()) (Var(v))
          (SVar(v,Knowledge.id_of(k),Gen)))] | K(_) when (Knowledge.derives
          k (asg) mm) ->  [(mm, Assignment.void())] | PN(_,_) when
          (Knowledge.derives k (asg) mm) ->  [(mm, Assignment.void())]
          | PNp(_,_) when (Knowledge.derives k (asg) mm) ->  [(mm,
          Assignment.void())] | PNm(_,_) when (Knowledge.derives k (asg) mm)
          ->  [(mm, Assignment.void())] | NO(_,_) when (Knowledge.derives k
          (asg) mm) ->  [(mm, Assignment.void())]

          | SVar(v,i,Gen) -> [(SVar(v,i,GenR), Assignment.add
          (Assignment.void()) (SVar(v,i,Gen)) (SVar(v,i,GenR)))] | SVar(v,i,t)
          -> [(SVar(v,i,t), (Assignment.void()))]

          | Publ(SVar(v,i,RName)) -> [(mm, (Assignment.void()))] |
          Priv(SVar(v,i,RName)) -> [(mm, (Assignment.void()))]

          | Couple(m1, m2) ->
              let res1 = mu kman k asg m1 in let resl = debox (map res1
              (function (m1,s1) -> let tt = l kman s1 (s1 m2) in map tt
              (function s2 -> (s2 m1, s2*s1)))) in let res2 = (map (map resl
              (snd)) (function sigma -> mu kman k (asg) (sigma m2))) in
              (rebuild resl res2 (function (m1,s1) -> function (m2,s2) ->
              (Couple(m1,m2), s1*s2)))

          | Crypt(m, K(s,i)) -> (
              let res =
                if (Knowledge.derives k (asg) (K(s,i))) then map (mu kman
                k asg m) (function (m1,s1) -> (Crypt(m1, K(s,i)), s1)) else []
              in let kl = (Knowledge.getCrypts k (K(s,i))) in res@(mulist
              kman k asg kl m)
             )
          | Crypt(m, PNp(s,p)) ->  (
              let res =
                if (Knowledge.derives k (asg) (PNm(s,p))) then map (mu kman k
                asg m) (function (m1,s1) -> (Crypt(m1, PNm(s,p) ), s1)) else []
              in let kl = (Knowledge.getCrypts k (PNm(s,p))) in res@(mulist
              kman k asg kl m)
             )
          | Crypt(m, PNm(s,p)) -> (
              let res =
                if (Knowledge.derives k (asg) (PNp(s,p))) then map (mu kman k
                asg m) (function (m1,s1) -> (Crypt(m1, PNp(s,p)), s1)) else []
              in let kl = (Knowledge.getCrypts k (PNp(s,p))) in res@(mulist
              kman k asg kl m)
```

```
             )

          | Crypt(m, SVar(v,i,SKey)) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, (SVar(v,i,SKey))),s1))
          | Crypt(m, Publ(SVar(v,i,RName))) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, Priv(SVar(v,i,RName))),s1))
          | Crypt(m, Priv(SVar(v,i,RName))) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, Publ(SVar(v,i,RName))),s1))

          | Crypt(m, SVar(v,i,PMKey)) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, SVar(v,i,MPKey)),s1))

          | Crypt(m, SVar(v,i,MPKey)) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, SVar(v,i,PMKey)),s1))

          | Crypt(m, SVar(v,i,t)) ->
              let res = mu kman k asg m in map res (function (m1,s1) ->
              (Crypt(m1, SVar(v,i,t)),s1))
          | _ -> [] (*map (l kman asg mm) (function s -> (s mm,s))*)
         )
and mulist =
  function kman ->
    function k ->
      function asg ->
        function cc ->
          function n ->
            match cc with | Crypt(m,kk)::ccs ->
                let rr = ni kman asg m n in (map rr (function r ->
                (Crypt(m,kk),r)))@(mulist kman k asg ccs n)
            | [] -> []
and ni =
  function kman ->
    function asg ->
      function m ->
        function n ->
          match m,n with | _ , BVar(v) -> [Assignment.add (Assignment.void())
          (Var(v)) m] | K(_), _ when n=m ->  [(Assignment.void())] |
          PN(_,_), _  when n=m ->  [(Assignment.void())] | PNp(_,_),
          _ when n=m ->  [(Assignment.void())] | PNm(_,_), _ when n=m ->
          [(Assignment.void())] | NO(_,_), _ when n=m ->  [(Assignment.void())]
          | Crypt(mm,K(s,i)), Crypt(nn,K(w,y)) when (s=w && i=y)-> ni kman
          asg mm nn | Crypt(mm,PNp(s1,p1)), Crypt(nn,PNm(s2,p2)) when
          (s1=s2 && p1=p2) -> ni kman asg mm nn | Crypt(mm,PNm(s1,p1)),
          Crypt(nn,PNp(s2,p2)) when (s1=s2 && p1=p2) -> ni kman asg mm nn |
          Crypt(mm, SVar(v1,i,SKey)), Crypt(nn, SVar(v2,j,SKey)) ->
              let min = if i<j then i else j in let ss = [Assignment.add
              (Assignment.add (Assignment.void()) (SVar(v1,i,SKey))
              (SVar(v1,min,SKey))) (SVar(v2,j,SKey)) (SVar(v1,min,SKey))]
              in let res = map ss (function sigma -> ni kman asg (sigma mm)
```

```
    (sigma nn)) in rebuild ss res (function (s1) -> function (s2) ->
    (s1*s2))

| Couple(mm1,mm2), Couple(nn1,nn2) ->
    let res1 = ni kman asg mm1 nn1 in let res2 = (map res1 (function
    sigma -> ni  kman (asg*sigma) (sigma mm2) (sigma nn2))) in
    rebuild res1 res2 (function (s1) -> function (s2) -> (s1*s2))
| SVar(v1,i,Gen), SVar(v2,j,Gen) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (SVar(v1,min,Gen))) n (SVar(v1,min,GenR))]

| SVar(v1,i,t1), SVar(v2,j,t2) when comparable(t1,t2) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (SVar(v1,min,mintype(t1,t2)))) n
    (SVar(v1,min,mintype(t1,t2)))]

| Publ(SVar(v1,i,t1)), Publ(SVar(v2,j,t2)) when comparable(t1,t2) ->
    let min = if i<j then i else j in [Assignment.add
    (Assignment.add (Assignment.void()) (SVar(v1,i,t1))
    (SVar(v1,min,mintype(t1,t2)))) (SVar(v2,j,t2))
    (SVar(v1,min,mintype(t1,t2)))]
| Priv(SVar(v1,i,t1)), Priv(SVar(v2,j,t2)) when comparable(t1,t2) ->
    let min = if i<j then i else j in [Assignment.add
    (Assignment.add (Assignment.void()) (SVar(v1,i,t1))
    (SVar(v1,min,mintype(t1,t2)))) (SVar(v2,j,t2))
    (SVar(v1,min,mintype(t1,t2)))]
| SVar(v1,i,t1), Publ(SVar(v2,j,t2)) when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (Publ(SVar(v1,min,t2)))) (SVar(v2,j,t2))
    ((SVar(v2,min,t2)))]
| SVar(v1,i,t1), Priv(SVar(v2,j,t2)) when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (Priv(SVar(v1,min,t2)))) (SVar(v2,j,t2))
    ((SVar(v2,min,t2)))]
| Publ(SVar(v2,j,t2)), SVar(v1,i,t1)  when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) (SVar(v2,j,t2)) ((SVar(v1,min,t2)))) n
    (Publ((SVar(v2,min,t2))))]
| Priv(SVar(v2,j,t2)), SVar(v1,i,t1)  when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) (SVar(v2,j,t2)) ((SVar(v1,min,t2)))) n
    (Priv((SVar(v2,min,t2))))]
| sm, SVar(v,i,Gen)  when notSVar(sm) ->
    let res = mu kman (Kmanager.getK kman i) asg (m) in map res
    (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
    n mm)))
| sm, SVar(v,i,GenR)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    Knowledge.knows oldk mm) in map res2 (function (mm,ss) ->
```

```
    (ss*Assignment.add (Assignment.void()) n mm))
| sm, SVar(v,i,SKey)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman oldk asg
    (m) in let res2 = filter res (function (mm,ss) -> ((isskey
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) n mm))
| sm, SVar(v,i,Name)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman oldk asg
    (m) in let res2 = filter res (function (mm,ss) -> ((isname
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) n mm))
| sm, SVar(v,i,RName)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    (Knowledge.isRName oldk mm )) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) n mm))
| sm, SVar(v,i,t)  when (notSVar(sm) && (t=MPKey ||  t=PMKey)) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    Knowledge.isPMKey oldk mm) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) n mm))

| SVar(v,i,Gen), sm  when notSVar(sm) ->
    let res = mu kman (Kmanager.getK kman i) asg (n) in map res
    (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
    m mm)))
| SVar(v,i,GenR), sm  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (n) in let res2 = filter res (function (mm,ss) ->
    Knowledge.knows oldk mm) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) m mm))
| SVar(v,i,SKey), sm  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman oldk asg
    (n) in let res2 = filter res (function (mm,ss) -> ((isskey
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) m mm))
| SVar(v,i,Name), sm  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman oldk asg
    (n) in let res2 = filter res (function (mm,ss) -> ((isname
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) m mm))
| SVar(v,i,RName), sm  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (n) in let res2 = filter res (function (mm,ss) ->
    (Knowledge.isRName oldk mm )) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) m mm))
| SVar(v,i,t), sm  when (notSVar(sm) && (t=MPKey ||  t=PMKey)) ->
    let oldk = (Kmanager.getK kman i) in let res = mu kman
    oldk asg (n) in let res2 = filter res (function (mm,ss) ->
    Knowledge.isPMKey oldk mm) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) m mm))
| SVar(v,i,_), sm when notSVar(sm) ->
    let res = mu kman  (Kmanager.getK kman i) asg (n) in map res
    (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
```

```
            m mm)))
        | _ -> []



(* symbolic unification functions *) let rec mu_eq =
  function kman ->
    function k ->
      function asg ->
        function mm -> (
          match mm with | Var(v) -> if (Knowledge.hasSKeys (Kmanager.getK kman
          1)) then [( SVar(v,1,SKey),  (Assignment.add (Assignment.void())
          mm (SVar(v,1,SKey)) ))] else [] | K(_) when (Knowledge.derives
          k (asg) mm) ->  [(mm, Assignment.void())] | PN(_,_) when
          (Knowledge.derives k (asg) mm) ->  [(mm, Assignment.void())]
          | PNp(_,_) when (Knowledge.derives k (asg) mm) ->  [(mm,
          Assignment.void())] | PNm(_,_) when (Knowledge.derives k (asg) mm)
          ->  [(mm, Assignment.void())] | NO(_,_) when (Knowledge.derives k
          (asg) mm) ->  [(mm, Assignment.void())] | SVar(v,i,Gen) -> [(mm,
          Assignment.void())] | SVar(v,i,GenR) -> [(mm, Assignment.void())]

          | SVar(v,i,Name) ->
              let oldk = Kmanager.getK kman i in let things =
              Knowledge.getAllNames oldk in let res = map things (function x ->
              (x, Assignment.add (Assignment.void()) (mm) x)) in res
          | SVar(v,i,RName) ->
              let oldk = Kmanager.getK kman i in let things =
              Knowledge.getRNames oldk in let res = map things (function x ->
              (x, Assignment.add (Assignment.void()) (mm) x)) in res
          | SVar(v,i,SKey) ->
              let oldk = Kmanager.getK kman i in let things =
              Knowledge.getSKeys oldk in let res = map things (function x ->
              (x, Assignment.add (Assignment.void()) (mm) x)) in res
          | SVar(v,i,MPKey) ->
              let oldk = Kmanager.getK kman i in let things =
              (Knowledge.getMPKeys oldk) in let res = map things (function x ->
              (x, Assignment.add (Assignment.void()) (mm) x)) in res
          | SVar(v,i,PMKey) ->
              let oldk = Kmanager.getK kman i in let things =
              (Knowledge.getPMKeys oldk) in let res = map things (function x ->
              (x, Assignment.add (Assignment.void()) (mm) x)) in res

          | Publ(SVar(v,i,RName)) ->
              let oldk = Kmanager.getK kman i in let things =
              (Knowledge.getRNames oldk) in let res = map things (function
              PN(pn,pi) -> (PNp(pn,pi), Assignment.add (Assignment.void())
              (SVar(v,i,RName)) (PN(pn,pi)) )) in res
          | Priv(SVar(v,i,RName)) ->
              let oldk = Kmanager.getK kman i in let things =
              (Knowledge.getRNames oldk) in let res = map things (function
              PN(pn,pi) -> (PNm(pn,pi), Assignment.add (Assignment.void())
              (SVar(v,i,RName)) (PN(pn,pi)) )) in res

          | Publ(SVar(v,i,Name)) ->
```

```
    let oldk = Kmanager.getK kman i in let things =
    (Knowledge.getAllNames oldk) in let res = map things (function
    PN(pn,pi) -> (PNp(pn,pi), Assignment.add (Assignment.void())
    (SVar(v,i,RName)) (PN(pn,pi)) )) in res
| Priv(SVar(v,i,Name)) ->
    let oldk = Kmanager.getK kman i in let things =
    (Knowledge.getAllNames oldk) in let res = map things (function
    PN(pn,pi) -> (PNm(pn,pi), Assignment.add (Assignment.void())
    (SVar(v,i,RName)) (PN(pn,pi)) )) in res

| Couple(m1, m2) ->
    let res1 = mu_eq kman k asg m1 in let res2 = (map (map res1
    (function (a,b) ->  b)) (function sigma -> mu_eq kman k (asg)
    (sigma m2))) in (rebuild res1 res2 (function (m1,s1) -> function
    (m2,s2) -> (Couple(m1,m2), s1*s2)))

| Crypt(m, K(s,i)) -> (
    let res =
      if (Knowledge.derives k (asg) (K(s,i))) then map (mu_eq kman
      k asg m) (function (m1,s1) -> (Crypt(m1, K(s,i)), s1)) else []
    in let kl = (Knowledge.getCrypts k (K(s,i))) in res@(mulist_eq
    kman k asg kl m)
   )
| Crypt(m, PNp(s,p)) ->  (
    let res =
      if (Knowledge.derives k (asg) (PNm(s,p))) then map (mu_eq
      kman k asg m) (function (m1,s1) -> (Crypt(m1, PNp(s,p) ),
      s1)) else []
    in let kl = (Knowledge.getCrypts k (PNm(s,p))) in res@(mulist_eq
    kman k asg kl m)
   )
| Crypt(m, PNm(s,p)) -> (
    let res =
      if (Knowledge.derives k (asg) (PNp(s,p))) then map (mu_eq
      kman k asg m) (function (m1,s1) -> (Crypt(m1, PNm(s,p)),
      s1)) else []
    in let kl = (Knowledge.getCrypts k (PNp(s,p))) in res@(mulist_eq
    kman k asg kl m)
   )

| Crypt(m, SVar(v,i,t)) ->
    let res = mu_eq kman k asg (SVar(v,i,t)) in let res2 = map res
    (function (a,b) -> mu_eq kman k asg (b m)) in rebuild res res2
    (function (a,b) -> function (c,d) -> (Crypt(c,a), b*d) )

| Crypt(m, Publ(SVar(v,i,t))) ->
    let res = mu_eq kman k asg (Publ(SVar(v,i,t))) in let res2 =
    map res (function (a,b) -> mu_eq kman k asg (b m)) in rebuild
    res res2 (function (a,b) -> function (c,d) -> (Crypt(c,a), b*d) )

| Crypt(m, Priv(SVar(v,i,t))) ->
    let res = mu_eq kman k asg (Priv(SVar(v,i,t))) in let res2 =
    map res (function (a,b) -> mu_eq kman k asg (b m)) in rebuild
    res res2 (function (a,b) -> function (c,d) -> (Crypt(c,a), b*d) )
```

```
          | _ -> []
         )
and mulist_eq =
  function kman ->
    function k ->
      function asg ->
        function cc ->
          function n ->
            match cc with | Crypt(m,kk)::ccs ->
                let rr = ni_eq kman asg m n in (map rr (function r ->
                (Crypt(m,kk),r)))@(mulist_eq kman k asg ccs n)
             | [] -> []
and ni_eq =
  function kman ->
    function asg ->
      function m ->
        function n ->
          match m,n with | K(_), _ when n=m ->  [(Assignment.void())]
          | PN(_,_), _  when n=m ->  [(Assignment.void())] | PNp(_,_),
          _ when n=m ->  [(Assignment.void())] | PNm(_,_), _ when n=m ->
          [(Assignment.void())] | NO(_,_), _ when n=m ->  [(Assignment.void())]
          | Crypt(mm,K(s,i)), Crypt(nn,K(w,y)) when (s=w && i=y) -> ni_eq
          kman asg mm nn | Crypt(mm,PNp(s1,p1)), Crypt(nn,PNp(s2,p2)) when
          (s1=s2 && p1=p2) -> ni_eq kman asg mm nn | Crypt(mm,PNm(s1,p1)),
          Crypt(nn,PNm(s2,p2)) when (s1=s2 && p1=p2) -> ni_eq kman asg mm nn |
          Crypt(mm, SVar(v1,i,t1)), Crypt(nn, SVar(v2,j,t2)) when (t1=SKey ||
          t1=MPKey || t1=PMKey)&&(t1=t2) ->
              let res = ni_eq kman asg (SVar(v1,i,t1)) (SVar(v2,j,t2)) in let
              res2 = map res (function sigma -> ni_eq kman asg (sigma mm)
              (sigma nn)) in rebuild res res2 (function (s1) -> function
              (s2) -> (s1*s2))

          | Couple(mm1,mm2), Couple(nn1,nn2) ->
              let res1 = ni_eq kman asg mm1 nn1 in let res2 = (map res1
              (function sigma -> ni_eq kman (asg) (sigma mm2) (sigma nn2)))
              in rebuild res1 res2 (function (s1) -> function (s2) -> (s1*s2))

          | Var(v1), Var(v2) ->
              [Assignment.add (Assignment.void()) m n]
          | Var(v1), _ ->
              [Assignment.add (Assignment.void()) m n]
          | _, Var(v2) ->
              [Assignment.add (Assignment.void()) n m]

          | SVar(v1,i,Gen), SVar(v2,j,Gen) ->
              if n=m then [Assignment.void()] else
                let min = if i<j then i else j in [Assignment.add
                (Assignment.add (Assignment.void()) m (SVar(v1,min,Gen)))
                n (SVar(v1,min,Gen))]
          | SVar(v1,i,GenR), SVar(v2,j,GenR) ->
              if n=m then [Assignment.void()] else
                let min = if i<j then i else j in [Assignment.add
                (Assignment.add (Assignment.void()) m (SVar(v1,min,GenR)))
```

```
      n (SVar(v1,min,GenR))]
| SVar(v1,i,Name), SVar(v2,j,Name) ->
    let min = if i<j then i else j in let oldk = Kmanager.getK kman
    min in let sk = Knowledge.getAllNames oldk in let ss = map sk
    (function x -> Assignment.add (Assignment.add (Assignment.void())
    (m) (x) ) (n) (x)) in ss
| SVar(v1,i,RName), SVar(v2,j,RName) ->
    let min = if i<j then i else j in let oldk = Kmanager.getK kman
    min in let sk = Knowledge.getRNames oldk in let ss = map sk
    (function x -> Assignment.add (Assignment.add (Assignment.void())
    (m) (x) ) (n) (x)) in ss
| SVar(v1,i,SKey), SVar(v2,j,SKey) ->
    let min = if i<j then i else j in let oldk = Kmanager.getK
    kman min in let sk = Knowledge.getSKeys oldk in let ss = map sk
    (function x -> Assignment.add (Assignment.add (Assignment.void())
    (m) (x) ) (n) (x)) in ss
| SVar(v1,i,MPKey), SVar(v2,j,MPKey) ->
    let min = if i<j then i else j in let oldk = Kmanager.getK kman
    min in let sk = Knowledge.getMPKeys oldk in let ss = map sk
    (function x -> Assignment.add (Assignment.add (Assignment.void())
    (m) (x) ) (n) (x)) in ss
| SVar(v1,i,PMKey), SVar(v2,j,PMKey) ->
    let min = if i<j then i else j in let oldk = Kmanager.getK kman
    min in let sk = Knowledge.getPMKeys oldk in let ss = map sk
    (function x -> Assignment.add (Assignment.add (Assignment.void())
    (m) (x) ) (n) (x)) in ss

| SVar(v1,i,t1), SVar(v2,j,t2) when comparable(t1,t2) ->
    let min = if i<j then i else j in let oldk = (Kmanager.getK kman
    min) in let mintg = mintype(t1,t2) in let tasg = Assignment.add
    (Assignment.add (Assignment.void()) m (SVar(v1,min,mintg)))
    n (SVar(v1,min,mintg)) in let mint = if min=i then t1
    else t2 in let minm = if mintg=t1 then SVar(v1,min,t1) else
    SVar(v2,min,t2) in let maxm = if mintg=t1 then SVar(v2,min,t2)
    else SVar(v1,min,t1) in let res = mu_eq kman oldk asg (minm)
    in map res (function (mm,ss) -> tasg*ss*(Assignment.add
    (Assignment.add (Assignment.void()) maxm mm ) minm mm))

| Publ(SVar(v1,i,t1)), Publ(SVar(v2,j,t2)) when comparable(t1,t2) ->
    let min = if i<j then i else j in [Assignment.add
    (Assignment.add (Assignment.void()) (SVar(v1,i,t1))
    (SVar(v1,min,mintype(t1,t2)))) (SVar(v2,j,t2))
    (SVar(v1,min,mintype(t1,t2)))]
| Priv(SVar(v1,i,t1)), Priv(SVar(v2,j,t2)) when comparable(t1,t2) ->
    let min = if i<j then i else j in [Assignment.add
    (Assignment.add (Assignment.void()) (SVar(v1,i,t1))
    (SVar(v1,min,mintype(t1,t2)))) (SVar(v2,j,t2))
    (SVar(v1,min,mintype(t1,t2)))]


| SVar(v1,i,t1), Publ(SVar(v2,j,t2)) when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (Publ(SVar(v1,min,t2)))) (SVar(v2,j,t2))
```

```
    ((SVar(v2,min,t2)))]
| SVar(v1,i,t1), Priv(SVar(v2,j,t2)) when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) m (Priv(SVar(v1,min,t2)))) (SVar(v2,j,t2))
    ((SVar(v2,min,t2)))]
| Publ(SVar(v2,j,t2)), SVar(v1,i,t1)  when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) (SVar(v2,j,t2)) ((SVar(v1,min,t2)))) n
    (Publ((SVar(v2,min,t2))))]
| Priv(SVar(v2,j,t2)), SVar(v1,i,t1)  when (t1=Gen ||
t1=GenR)&&(t2=Name || t2=RName) ->
    let min = if i<j then i else j in [Assignment.add (Assignment.add
    (Assignment.void()) (SVar(v2,j,t2)) ((SVar(v1,min,t2)))) n
    (Priv((SVar(v2,min,t2))))]

| sm, SVar(v,i,Gen)  when notSVar(sm) ->
    let res = mu_eq kman (Kmanager.getK kman i) asg (m) in map res
    (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
    n mm)))
| sm, SVar(v,i,GenR)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    Knowledge.knows oldk  mm) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) n mm))
| sm, SVar(v,i,SKey)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman oldk
    asg (m) in let res2 = filter res (function (mm,ss) -> ((isskey
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) n mm))
| sm, SVar(v,i,Name)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman oldk
    asg (m) in let res2 = filter res (function (mm,ss) -> ((isname
    mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
    (Assignment.void()) n mm))
| sm, SVar(v,i,RName)  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    (Knowledge.isRName oldk mm )) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) n mm))
| sm, SVar(v,i,t)  when (notSVar(sm) && (t=MPKey ||  t=PMKey)) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
    oldk asg (m) in let res2 = filter res (function (mm,ss) ->
    Knowledge.isPMKey oldk mm) in map res2 (function (mm,ss) ->
    (ss*Assignment.add (Assignment.void()) n mm))

| SVar(v,i,Gen), sm  when notSVar(sm) ->
    let res = mu_eq kman (Kmanager.getK kman i) asg (n) in map res
    (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
    m mm)))
| SVar(v,i,GenR), sm  when notSVar(sm) ->
    let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
    oldk asg (n) in let res2 = filter res (function (mm,ss) ->
```

```
            Knowledge.knows oldk mm) in map res2 (function (mm,ss) ->
            (ss*Assignment.add (Assignment.void()) m mm))
          | SVar(v,i,SKey), sm  when notSVar(sm) ->
            let oldk = (Kmanager.getK kman i) in let res = mu_eq kman oldk
            asg (n) in let res2 = filter res (function (mm,ss) -> ((isskey
            mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
            (Assignment.void()) m mm))
          | SVar(v,i,Name), sm  when notSVar(sm) ->
            let oldk = (Kmanager.getK kman i) in let res = mu_eq kman oldk
            asg (n) in let res2 = filter res (function (mm,ss) -> ((isname
            mm))) in map res2 (function (mm,ss) -> (ss*Assignment.add
            (Assignment.void()) m mm))
          | SVar(v,i,RName), sm  when notSVar(sm) ->
            let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
            oldk asg (n) in let res2 = filter res (function (mm,ss) ->
            (Knowledge.isRName oldk mm )) in map res2 (function (mm,ss) ->
            (ss*Assignment.add (Assignment.void()) m mm))
          | SVar(v,i,t), sm  when (notSVar(sm) && (t=MPKey ||  t=PMKey)) ->
            let oldk = (Kmanager.getK kman i) in let res = mu_eq kman
            oldk asg (n) in let res2 = filter res (function (mm,ss) ->
            Knowledge.isPMKey oldk mm) in map res2 (function (mm,ss) ->
            (ss*Assignment.add (Assignment.void()) m mm))
          | SVar(v,i,_), sm when notSVar(sm) ->
            let res = mu_eq kman  (Kmanager.getK kman i) asg (n) in map res
            (function (mm,ss) -> (ss*(Assignment.add (Assignment.void())
            m mm)))
          | _ -> []
```

Csolver

```
open Utils open Message


type cns = PC of (msg*msg) | NC of (msg*msg) type cs = cns list list

let init =
  function () -> [[]]


let expandable =
  function pc1 ->
    function pc2 ->
      match pc1,pc2 with | PC(v1,m1), PC(v2,m2) when m2=v1 -> true |
      PC(v1,m1), NC(v2,m2) when m2=v1 -> true | NC(v1,m1), PC(v2,m2) when
      m2=v1 -> true | NC(v1,m1), NC(v2,m2) when m2=v1 -> true | _ , _ -> false

let expand =
  function pcte ->
    function pc ->
```

```
      match pcte, pc with | PC(v1,m1), PC(v2,m2) when v2=m1 -> PC(v1,m2) |
      PC(v1,m1), NC(v2,m2) when v2=m1 -> NC(v1,m2) | NC(v1,m1), PC(v2,m2)
      when v2=m1 -> NC(v1,m2) | NC(v1,m1), NC(v2,m2) when v2=m1 -> PC(v1,m2)
      | _ , _ -> pcte


let rec closure =
  function cse ->
    function pc ->
      let ltr = filter cse (expandable pc) in let rtl = filter cse ((invert
      expandable) pc) in let toadd1 = map ltr (function x -> expand x pc)
      in let toadd2 = map rtl (function x -> expand pc x) in toadd1@toadd2


let rec addc =
  function cse ->
    function pcc ->
      match pcc with | PC(v1,m1) when v1=m1 -> cse | NC(v1,m1) when v1=m1 ->
      cse | _ ->
          if exists cse (function x -> x=pcc) then cse else
            let toadd = closure cse pcc in if toadd = [] then (pcc::cse)
            else fold toadd (invert (addc) ) (pcc::cse)


let addconstraint =
  function pc ->
    function cs ->
      map cs ((invert addc) pc)

let addconstraints =
  function pcl ->
    function cs ->
      if pcl = [] then cs else fold pcl (addconstraint) cs

let adddisjunct =
  function cs ->
    function pcl ->
      mul cs pcl (addc)

let adddisjuncts =
  function pcl ->
    function cs ->
      if pcl = [] then cs else fold pcl (invert adddisjunct) cs

let addconjuncts =
  function pcl ->
    function cs ->
      if pcl = [] then cs else debox(mul [cs] pcl (adddisjunct))


let toasg =
  function cs ->
    map cs ( function cse ->
      let pcs = filter cse (function x -> match x with | PC(_,_) -> true |
```

```
      _ -> false) in let table = map pcs (function PC(a,b) ->  (a,b)) in
      Assignment.newasg table
   )


let preserve_truth =
  function cs ->
    let rec checkcse =
      function cse ->
        function cso ->
          match cse with | PC(v,m)::css ->
              if exists cso (function x -> x=NC(v,m)) then false else
                if exists cso (
                  function x ->
                    match x with | PC(vv,mm) -> v=vv && mm<>m && (notSVar m &&
                    notSVar mm && notVar m && notVar mm) | _ -> false )
                then false else checkcse css cso
          | NC(v,m)::css ->
              if exists cso (function x -> x=PC(v,m)) then false else checkcse
              css cso
          | [] -> true
    in filter cs (half (checkcse))

let check =
        function cs ->
                cs = []

let positivize =
  function pcl ->
    map pcl (function x -> PC(x))

let negativize =
  function pcl ->
    map pcl (function x -> NC(x))


let linearize =
  function cs ->
    map (cs) (function x -> [x])


let sss = function l1 -> function l2 -> let cff = function
v1 -> function v2 -> match v1,v2 with | Var(nv1,(np1,ii1)),
Var(nv2,(np2,ii2)) -> (string_of_int(ii1)^nv1) > (string_of_int(ii2)^nv2) |
Var(nv1,(np1,ii1)), SVar((nv2,(np2,ii2)),_,_) -> (string_of_int(ii1)^nv1) >
(string_of_int(ii2)^nv2) | SVar((nv1,(np1,ii1)),_,_), Var(nv2,(np2,ii2))
-> (string_of_int(ii1)^nv1) > (string_of_int(ii2)^nv2) |
SVar((nv1,(np1,ii1)),_,_), SVar((nv2,(np2,ii2)),_,_) ->
(string_of_int(ii1)^nv1) > (string_of_int(ii2)^nv2)

in

match l1, l2 with | PC(v1,m1),PC(v2,m2) -> cff v1 v2 | PC(v1,m1),NC(v2,m2)
-> true | NC(v1,m1),PC(v2,m2) -> false | NC(v1,m1),NC(v2,m2) -> cff v1 v2
```

```
let prnt =
  function cs ->
    print_list cs ( function cse ->
             let scse = sort cse (sss) in
      print_list scse ( function cc ->
        match cc with | PC(Var(v),m1) -> prntMsg (Var(v)); print_string("
        -> "); prntMsg m1; print_string("\n") | NC(Var(v),m1) -> prntMsg
        (Var(v)); print_string(" \\> "); prntMsg m1; print_string("\n")

        | PC(n1,m1) -> prntMsg (n1); print_string(" -> "); prntMsg m1;
        print_string("\n") | NC(n1,m1) -> prntMsg (n1); print_string(" \\>
        "); prntMsg m1; print_string("\n")

        | _ -> ()
     ); print_string("\n")
   )

let prntHtml =
  function cs ->
    print_list cs ( function cse ->
             let scse = sort cse (sss) in
      print_list scse ( function cc ->
        match cc with

        | PC(Var(v),m1) -> prntMsgHtml (Var(v)); print_string(" -&gt ");
        prntMsgHtml m1; print_string("<br>") | NC(Var(v),m1) -> prntMsgHtml
        (Var(v)); print_string(" \\&gt "); prntMsgHtml m1; print_string("<br>")

        | PC(n1,m1) -> prntMsgHtml (n1); print_string(" -&gt ");
        prntMsgHtml m1; print_string("<br>") | NC(n1,m1) -> prntMsgHtml
        (n1); print_string(" \\&gt "); prntMsgHtml m1; print_string("<br>")

        | _ -> ()
     ); print_string("<br>")
   )
```

Gviz

```
open Node open Parsecmdline


let initgraph =
  function s ->
    print_string("\n/*automatically generated*/\ndigraph
    "^s^"{\nconcentrate=true\n ")

let donegraph =
  function () ->
    print_string("\n}\n/*end of graph*/\n")
```

```
let rec prntTrace =
  function n ->
    function go ->
      let (ct,asg,kb) = Node.getstatus n in let id = Node.getid n in let fth =
      Node.getfather n in let level = Node.getlevel n in let nodeinfo =
        function () ->
          print_string("\n"); print_string(string_of_int(id)^"[label=\"");

          if go.actions then (
            print_string("Action: "); let act = Node.getlabel n in
            print_string((Action.tostring act)^"\\n\\n");
           ) else print_string(" ");

          if go.context then (
            print_string("Context:\\n------------\\n");
            print_string(Context.tostring ct asg);
           ) else ();

          if go.k then (
            print_string("Knowledge:\\n------------\\n");
            print_string(Knowledge.tostring kb);
           ) else ();

          if go.asg then (
            print_string("Assignment:\\n------------\\n");
            print_string(Assignment.tostring asg);
           ) else ();

          print_string("\"]\n"); print_string(string_of_int(id)^" ");
      in (
       if Node.isNul fth then (
         print_string(string_of_int(id)); nodeinfo();
         print_string(string_of_int(id)^" ")
        )
       else
         (
          prntTrace fth go; if (level<go.level ) then (
            print_string("-> "^string_of_int(id)); nodeinfo()
           )
          else ()

         )
      )


let rec prntTrace2 =
  function n ->
    function bm ->
      function go ->
        let (ct,asg,kb) = Node.getstatus n in let id = Node.getid n in let
        fth = Node.getfather n in let nodeinfo =
          function () ->
            print_string("\n"); print_string(string_of_int(id)^"[label=\"");
```

```
            if go.context then (
              print_string("Context:\\n------------\\n");
              print_string(Context.tostring ct asg);
             ) else ();

            if go.k then (
              print_string("Knowledge:\\n------------\\n");
              print_string(Knowledge.tostring kb);
             ) else ();

            if go.asg then (
              print_string("Assignment:\\n------------\\n");
              print_string(Assignment.tostring asg);
             ) else ();

            print_string("\"]\n"); print_string(string_of_int(id)^" ");
        in (
         if Node.isNul fth then (
           print_string(string_of_int(id)); nodeinfo();
           print_string(string_of_int(id)^" ")
          )
         else
           (
            let fid = (Node.getid fth) in ( (
             if (bm.(fid)) then print_string(string_of_int(fid)^" ") else
             prntTrace2 fth bm go
            );

            print_string("-> "^string_of_int(id)); bm.(fid) <- true;

            if go.actions then (
              print_string("[label=\""); let act = Node.getlabel n in
              print_string(Action.tostring act); print_string("\"] ")
             ) else print_string(" ");

            nodeinfo()
           ) )
        )

let rec prntTree =
  function nls ->
    function nn ->
      function go ->
        let bitmap = Array.make nn (false) in let rec prntTree2 =
          function nl ->
            match nl with | n::ns -> prntTrace2 n bitmap go;
            print_string("\n"); prntTree2 ns | [] -> print_string("\n")
        in prntTree2 nls
```

Join

```
open Node open Message open Assignment open Action open Princpool open Utils


let rec iter =
  function i ->
    function j ->
      function f ->
        function en ->
          if i>j then en else  (f i) (iter (i+1) j f en)

let rec sum =
  function i ->
    function j ->
      function f ->
        iter i j ( compose ( f ) ( + ) ) 0

let rec cardfk =
  function n ->
    function k->
      function i ->
        if n=2 then 1 else sum 0 (k-1) (cardfk (n-1) (k-i))


let rec getinitialnodes =
  function n ->
    function k->
      if n=2 then k+1 else sum 0 k (getinitialnodes (n-1))


let initjoin =
  function pp ->
    function kman ->
      function thelist ->
        function node ->
          let nj = Node.getnj node in let k = Princpool.getmax pp in let
          rec adder =
            function nn ->
              function cl ->
                function ii ->
                  match cl with | x::xs -> (
                      let (oldct, oldasg, oldkb) = Node.getstatus nn
                      in let p = Princpool.getprinc pp (x-1) ii in
                      let name = Principal.getName p in let newkb =
                      (Kmanager.expandK(kman, oldkb, Assignment.void(),
                      Couple(PN(fst(name),snd(name)),PNp(fst(name),snd(name)))
                      )) in let newct = Context.add(oldct,p) in let newst =
                      Status.create(newct, oldasg, newkb) in if ii=1 then (
                        let newnode = Node.create(newst) in (
                        Node.setfather(newnode, node); Node.setlabel(newnode,
                        Join(name, Assignment.void())); adder newnode xs (ii+1)
                       ) )
```

```
            else (
              Node.setstatus(nn,newst); adder nn xs (ii+1)
             )
           )
        | [] -> nn
in let rec decodenj =
  function a ->
    function b ->
      let c = sum 0 (k) (cardfk (b) (k)) in if a-c <0 then (b, a)
      else decodenj (a-c) (b+1)
in let rec gen =
  function a ->
    function l ->
      if l = 0 then [] else a::(gen a (l-1))
in let rec generatelist =
  function npp ->
    function kk ->
      if npp = 1 then (
        [gen 1 kk]
       )
      else (
        if npp = 2 then (
          let tmp = ref(-1) in let base = gen 0 (kk+1) in let ttt=
          map base (function x -> (tmp:=!tmp+1; !tmp)) in map ttt
          (function x -> (gen 1 (kk-x))@(gen 2 x))
         )
        else
          let part = iter 0 kk (compose (generatelist (npp-1))
          ( @ )) [] in map part ( function l ->
            let sz = len l in l@(gen npp (kk-sz))
         )
       )
in let rec onecnt =
  function l ->
    function ix -> match l with | r::rs -> (gen ix r)@(onecnt rs
    (ix+1)) | [] -> []
in if nj= -1 then Node.nulNode() else
  let (oldct, oldasg, oldkb) = Node.getstatus node in if
  (Context.howmany oldct=k) then (
    Node.setnj(node,-1); Node.nulNode()
   ) else (
    if (thelist = []) then (
      let codelist = (generatelist (howmany pp) k) in let flag =
      (nj >= (len codelist)) in if flag then (
        Node.setnj(node,-1); Node.nulNode()
       ) else (
        let sel = ( select codelist nj) in ( Node.setnj(node,nj+1);
        adder node sel 1
       ) )
     ) else (
      Node.setnj(node,-1); adder node (onecnt thelist 1) 1;
     )
   )
```

```
let rec newAsg =
  function ls ->
    function ks ->
      match ls with | Var(v)::vs -> Assignment.add (newAsg vs ks) (Var(v))
      (K("k!"^ks,0)) | [] -> Assignment.void()
```

```
let joincontext =
  function kman ->
    function greed ->
      function node ->
        let (ct,asg,kb) = Node.getstatus node in let newlct = Context.getlist
        (Context.applyall ct asg) in let kbid = Knowledge.id_of kb in let
        rec closeprincipal =
          function pp ->
            match pp with | (a,b,v::vs,c) ->
                let tasg =
                  if (Principal.typeOf pp v) <> SKey then (Assignment.add
                  (Assignment.void()) (Var(v)) (SVar(v,kbid,Name))) else
                  Assignment.void()
                in tasg*(closeprincipal (a,b,vs,c))
            | (_,_,[],_) -> (Assignment.void())
        in let rec closep =
          function prs ->
            match prs with | pp::pps ->
                let newasg = closeprincipal pp in newasg*(closep pps)
            | [] -> (Assignment.void())
        in let rec closeprincipalgreed =
          function pp ->
            match pp with | (a,b,v::vs,c) ->
                let tasg =
                  if (Principal.typeOf pp v) <> SKey then (
                    let names = Knowledge.getAllNames kb in map names (function
                    n -> (Assignment.add (Assignment.void()) (Var(v)) (n)))
                   )
                  else [Assignment.void()]
                in mul tasg (closeprincipalgreed (a,b,vs,c)) (Assignment.comp)
            | (_,_,[],_) -> [(Assignment.void())]
        in let rec closepgreed =
          function prs ->
            match prs with | pp::pps ->
                let newasg = closeprincipalgreed pp in mul newasg (closepgreed
                pps) (Assignment.comp)
            | [] -> [Assignment.void()]
        in

        let rec instantiateKeys =
          function part ->
            match part with | x::xs ->
                let nk = ref(0) in let nasg = fold x (function ls -> function
                fasg -> nk:= !nk+1; ((newAsg ls (string_of_int(!nk)))*fasg))
```

```
                (Assignment.void()) in nasg::(instantiateKeys xs )
            | [] -> []
        in let ctasg =
          if (greed) then (
            let tempasg = (closepgreed newlct) in let newctg =
            Context.applyall ct (head tempasg) in let (nk,vars) =
            (Context.getOpenVariables newctg) in let parts = partition
            (vars@(Knowledge.getIntruderVariables kb)) in let asgres
            = instantiateKeys parts in let ctasgg = if (asgres = [])
            then [Assignment.void()] else asgres in mul tempasg ctasgg
            (Assignment.comp)
           )
          else(
            let tempasg =  (closep newlct) in let newct = Context.applyall
            ct tempasg in let (nk,vars) = (Context.getOpenVariables newct)
            in let parts = partition (vars@(Knowledge.getIntruderVariables
            kb)) in let asgres = instantiateKeys parts in if (asgres =
            []) then [tempasg] else map asgres (Assignment.comp tempasg)

           )
        in let allnewct = map ctasg (function sigma -> Context.applyall
        ct (asg*sigma)) in let allnewkb = map ctasg (function sigma ->
        Kmanager.updateK (kman, kb, sigma)) in let allnewst = couplify
        (couplify ctasg allnewct) allnewkb in let allnewnode = map allnewst (
          function ((newasg,newct),newkb) ->
            let newnode = Node.create(Status.create(newct, asg*newasg, newkb)
            ) in ( Node.setfather (newnode, node); Node.setlabel (newnode,
            Join(("Close Principals",Node.getid newnode),asg*newasg)); newnode
           )
         ) in allnewnode


let joinwithmodel =
  function kman ->
    function greed -> function node ->
      function mdls ->
        if map mdls (fst) = [] then ([]) else
          let (ct,asg,kb) = Node.getstatus node in debox ( map mdls ( function
          (ss,cs) ->
            let newasg = asg*ss in let newst = Status.create(ct, newasg, kb)
            in let newnode = Node.create(newst) in ( Node.setfather (newnode,
            node); Node.setlabel (newnode, Join(("Check Component",Node.getid
            newnode),ss)); map (joincontext kman greed newnode) (function
            nnn -> (nnn,cs))
           )
         ) )
```

Khash

```
open Knowledge

type khash = (kb Weak.t) array

let rec clear =
  function hsh ->
    function n ->
      if n= Array.length hsh then () else (Array.set hsh n (Weak.create 1);
      clear hsh (n+1))

let hash =
  function key ->
    function hsh ->
      let (f,i) = modf(0.6180339887 *. float_of_int(key)) in int_of_float(
      f *. float_of_int(Array.length hsh))

let init =
  function size ->
    let bkt = Weak.create 1 in let hsh = Array.make (min size
    (Sys.max_array_length-1)) bkt in (clear hsh 0; hsh)


let rec insertbkt =
  function bkt ->
    function kb ->
      function n ->
        if n = Weak.length bkt then (
          let nbkt = Weak.create (n+1) in Weak.set nbkt n (Some(kb));
          Weak.blit bkt 0 nbkt 0 n; nbkt
         )
        else (
          if (Weak.get bkt n = None) then (Weak.set bkt n (Some(kb)); bkt)
          else insertbkt bkt kb (n+1)
         )


let rec getbkt =
  function bkt ->
    function key ->
      function n ->
        if n = Weak.length bkt then ( Knowledge.void()) else (
          match Weak.get bkt n with | Some(kb) -> if (Knowledge.id_of kb =
          key) then kb else getbkt bkt key (n+1) | None -> getbkt bkt key (n+1)
         )


let insert =
  function hsh ->
    function kb ->
      let key = Knowledge.id_of kb in let h = hash key hsh in let bkt =
      Array.get hsh h in let nbkt = insertbkt bkt kb 0 in Array.set hsh h nbkt
```

```
let get =
  function hsh ->
    function key ->
      let h = hash key hsh in let bkt = Array.get hsh h in getbkt bkt key 0


let size =
  function hsh ->
    Array.fold_right (function bkt -> function mm -> max (Weak.length bkt)
    mm) hsh 0
```

Kmanager

```
open Message open Assignment open Knowledge open Khash


exception NoKnowledge

type km = khash*(int ref)

let initKM =
  function n -> (Khash.init n, ref(-1))

let getK =
  function (kman,mid) ->
    function n ->
      let kb = Khash.get kman n in if Knowledge.isVoid kb then
      (print_string(string_of_int(n)^"\n"); raise NoKnowledge) else kb

let newK =
  function (kman,mid) ->
    let newk = Knowledge.expand (Knowledge.void()) (!mid+1) (Assignment.void())
    in (
     Khash.insert kman newk; mid:=!mid+1; newk
    )

let expandK =
  function ((kman,mid),k,asg,m) ->
    let newk = Knowledge.applyall (Knowledge.learn ((Knowledge.expand (k)
    (!mid+1) asg),m)) asg in (
     Khash.insert kman newk; mid:=!mid+1; newk;
    )

let updateK =
  function (kmanage,k,asg) ->
    expandK (kmanage,k,asg,Bottom)
```

```
let size =
  function (kman,mid) -> Khash.size kman
```

Knowledge

```
open Message open Utils


type  kb = int*(msg list)


let void =
  function () -> (-1,[])


let applyall =
  function (kbid,k) ->
    function asg -> (kbid, map k (asg))

let expand =
  function (kbid,a) ->
    function n ->
      function asg ->
        let (ll, nk) =  applyall (kbid,a) asg in (n,nk)

let get =
  function (kbid,k) ->
    function f -> (filter k f)

let rec scan =
  function (kbid,k) ->
    function p -> exists k p

let knows =
  function k ->
      function m ->
        scan k (function mm ->  mm = m)

let knowsd =
  function k ->
    function asg ->
      function m ->
        scan k (function mm ->  (asg mm) = m)

let add =
  function ((kbid,k), n) -> (kbid, n::k)

let addlist =
  function ((kbid,k), n) -> (kbid, n@k)
```

```
let rec getlist =
  function (kbid, k) -> k

let rec getAllCrypts =
  function k ->
    get k (iscrypt)


let rec listfy =
  function (k,ln) ->
    let rec llistfy =
      function n ->
        (
         match n with | Couple(m1, m2) -> (llistfy m1)@(llistfy m2) |
         Crypt(mm, K(s,i)) when knows k (K(s,i))  -> llistfy mm | Crypt(mm,
         PNp(s,p)) when knows k  (PNm(s,p))  -> if (knows k (PNp(s,p)))
         then llistfy mm else n::(llistfy mm) | Crypt(mm, PNm(s,p)) when
         knows k  (PNp(s,p))  -> if (knows k (PNm(s,p))) then llistfy mm else
         n::(llistfy mm) | Crypt(mm, SVar(_,_,_)) -> llistfy mm | Crypt(mm,
         Publ(SVar(_,_,_))) -> llistfy mm | Crypt(mm, Priv(SVar(_,_,_)))
         -> llistfy mm | Crypt(mm,kk) when not(knows k (n)) -> [n] | K(_)
         when not (knows k n) -> [n] | PNp(s,p) when not (knows k n) -> [n] |
         PNm(s,p) when not (knows k n) -> [n] | NO(s,p) when not (knows k n)
         -> [n] | PN(s,p) when not (knows k n) -> [n] | SVar(_,_,_) when not
         (knows k n) -> [n] | Var(_) when not (knows k n) -> [n] | _ -> []
        ) in
    match ln with | m1::ms -> (llistfy(m1))@(listfy(k,ms)) | [] -> []

let rec learn =
  function (k,n) ->
    if n = Bottom then k else ( let gg = remclone (listfy(k,n::(getAllCrypts
    k))) (function m1 -> function m2 -> m1<>m2) in let kk = filter gg
    (function m -> (match m with | Crypt(_,_) -> false | _ -> true)&&(not
    (knows k m))) in let cc = filter gg (function m -> (match m with |
    Crypt(_,_) -> true  | _ -> false)&&(not (knows k m))) in if kk = []
    then addlist(k,cc) else learn (addlist(k,kk),n)
   )

let rec derives =
  function k ->
    function asg ->
      function m ->
        (match m with | Couple(m1,m2) ->  (derives k (asg) m1) && (derives k
        (asg) m2) | Crypt(m1,K(s,i)) -> (derives k (asg) m1) && (derives k
        (asg) (K(s,i))) | Crypt(m1,PNp(a,b)) -> (derives k (asg) m1) &&
        (derives k (asg) (PNm(a,b))) | Crypt(m1,PNm(a,b)) -> (derives k
        (asg) m1) && (derives k (asg) (PNp(a,b))) | SVar(v,i,_) -> true |
        o -> knowsd k asg o )||(knowsd k asg m)



let getIntruderVariables =
  function k ->
    get k (function m -> match m with | Var(_) -> true | _ -> false)
```

```
let rec getAllNames =
  function k ->
    get k (isname)

let rec getRNames =
  function k ->
      get k (function m -> (isname m) && (knows k (toPNp m)) && (knows k
      (toPNm m)))

let isRName =
  function k ->
    function n ->
      let rn = getRNames k in exists rn (function m -> m=n)



let rec getNames =
  function k ->
      diff (getAllNames k) (getRNames k)


let rec getKeys =
  function k ->
    get k (function m -> isskey m || ismkey m || ispkey m )

let rec getSKeys =
  function k ->
    get k isskey


let rec getAllPKeys =
  function k ->
    get k ispkey

let rec getPMKeys =
  function k ->
    get k (function m ->  (ispkey m && (knows k (compl(m)))))


let rec getPKeys =
  function k ->
    diff (getAllPKeys k) (getPMKeys k)


let isPMKey =
  function k ->
    function n ->
      let rn = getPMKeys k in exists rn (function m -> toName(m)=toName(n))

let rec getAllMKeys =
  function k ->
      get k ismkey
```

```
let rec getMPKeys =
  function k ->
      map (getPMKeys k) (function m -> compl(m))

let rec getMKeys =
  function k ->
      diff (getAllMKeys k) (getMPKeys k)


let rec getCrypts =
  function k ->
    function kk ->
      if (knows k  (compl(kk))) && (knows k  kk) then [] else (
        get k (function m ->
          (match m with | Crypt(_,ss) when (ss=kk)-> true | _ -> false )
              )
       )

let rec getSCrypts =
  function k ->
      let keys = getSKeys k in let res = map keys (function kk -> getCrypts
      k  kk) in debox res

let rec getPCrypts =
  function k ->
      let keys = getPKeys k in let res = map keys (function kp -> getCrypts
      k kp) in debox res

let rec getMCrypts =
  function k ->
      let keys = getMKeys k in let res = map keys (function km -> getCrypts
      k km) in debox res


let id_of =
  function (kbid,k) -> kbid


let isEmpty =
  function (kbid, k) -> k=[]

let isVoid =
  function (kbid, k) -> kbid= -1


let rec hasKeys =
  function  k ->
    scan k (function m -> (isskey m) || (ispkey m) || (ismkey m))

let rec hasSKeys =
  function  k ->
    scan k isskey
```

```
let rec hasPKeys =
  function  k ->
    scan k ispkey

let rec hasMKeys =
  function  k ->
    scan k ismkey

let rec hasNames =
  function  k ->
    scan k isname

let rec hasPMKeys =
  function  k ->
    scan k ( function m ->
      if (ispkey m || ismkey m) then (
        scan k (function mm -> mm = compl(m))
       ) else false
   )

let rec hasRNames =
  function  k ->
    scan k ( function m -> if isname m then (
      ((get k (function m1 -> m1 = toPNp m))<>[])&&((get k (function m1 ->
      m1 = toPNm m))<>[])
     ) else false
   )



let rec prntHtml =
  function (kbid, k) ->
    let r = ref(0) in let rec prntKb =
      function ml ->
        match ml with | m::xs -> Message.prntMsgHtml m; print_string(",
        "); r:=!r+1; (if (!r mod 5=0) then print_string("<br>") else ());
        prntKb xs | [] -> (print_string(".<br>"))
    in (prntKb k)


let rec prnt =
  function (kbid, k) ->
    let rec prntKb =
      function ml ->
        match ml with | m::xs -> Message.prntMsg m; print_string("\n");
        prntKb xs | [] -> (print_string(".\n"))
    in (print_string("Kid: "^string_of_int(kbid)^"\n"); prntKb k)

let rec tostring =
  function (kbid,k) ->
    let rec prntKb =
      function ml ->
        match ml with | m::[] -> (Message.tostring m)^(prntKb []) | m::xs ->
        (Message.tostring m)^(", ")^(prntKb xs) | [] -> (".\\n\\n")
```

```
    in ("("^string_of_int(kbid)^")"^prntKb k)
```

Logic

```
open Message open Node open Utils open Assignment


type index = string

type pls =
  | Forall of index*string*pls | Exists of index*string*pls | Not of pls |
  And of pls*pls | Or of pls*pls | Equal of msg*msg | Diff of msg*msg |
  Derive of msg | NDerive of msg | True | False


type npls =
    pls list list

let rec not_norm =
  function w ->
    match w with | Not(And(a,b)) -> Or(not_norm(Not(a)), not_norm(Not(b))) |
    Not(Or(a,b)) -> And(not_norm(Not(a)), not_norm(Not(b))) | Not(Not(a))
    -> not_norm a | Not(Equal(v,m)) -> Diff(v,m) | Not(Diff(v,m)) ->
    Equal(v,m) | Not(Derive(m)) -> NDerive(m) | Not(NDerive(m)) -> Derive(m)
    | Not(Forall(i,s,a)) -> Exists(i,s,not_norm(Not(a))) | Not(Exists(i,s,a))
    -> Forall(i,s,not_norm(Not(a))) | Not(True) -> False | Not(False) -> True
    | And(a,b) -> And(not_norm a, not_norm b) | Or(a,b) -> Or(not_norm a,
    not_norm b) | Forall(i,s,a) -> Forall(i,s, not_norm a) | Exists(i,s,a)
    -> Exists(i,s, not_norm a) | _ -> w

let  and_or_norm =
  function p ->
    let i = ref(0) in let rec and_or =
      function w ->
        match w with | And(Or(b,c),a) | And(a,Or(b,c)) ->
            let z = and_or (And(a,b)) in let v = and_or(And(a,c)) in (i:=
            !i+1; Or(z,v))
```

```
        | And(a,b) ->
            let v1 = !i in let c = and_or a in let v2 = !i in let d = and_or
            b in if (v1 = v2 && v2 = !i) then And(c,d) else and_or (And(c,d))
        | Or(a,b) -> Or(and_or a, and_or b) | _ -> w
    in and_or p



let rec expand_quantifiers =
  function w ->
    function ct ->
      let rec expand =
        function is ->
          function i ->
            function name ->
              function l ->
                let rec ist =
                  function mm ->
                    match mm with | Var(n,(p,ii)) when p=is -> Var(n,(name,i))
                    | Publ(Var(n,(p,ii))) when p=is -> Publ(Var(n,(name,i)))
                    | Priv(Var(n,(p,ii))) when p=is -> Priv(Var(n,(name,i)))
                    | NO(n,ii) when n.[0]='_' ->
                        let pll = (String.rindex n '_' ) in let nl
                        = String.length n in let iii = (String.sub n 1
                        (pll-1)) in if iii=is then NO(String.sub n (pll+1)
                        ((nl)-pll-1), i) else mm
                    | K(n,ii) when n.[0]='_' ->
                        let pll = (String.rindex n '_' ) in let nl
                        = String.length n in let iii = (String.sub n 1
                        (pll-1)) in if iii=is then K(String.sub n (pll+1)
                        ((nl)-pll-1), i) else mm
                    | PN(n,ii) when n=(name^is) -> PN(name,i) | PNm(n,ii) when
                    n=(name^is) -> PNm(name,i) | PNp(n,ii) when n=(name^is)
                    -> PNp(name,i) | Couple(m1,m2) -> Couple(ist m1, ist m2)
                    | Crypt(m1,kk) -> Crypt(ist m1, ist kk) | _ -> mm
                in match l with | And(a,b) -> And(expand is i name a,
                expand is i name b) | Or(a,b) -> Or(expand is i name a,
                expand is i name b) | Forall(ii,s,a) -> Forall(ii, s, expand
                is i name a) | Exists(ii,s,a) -> Exists(ii, s, expand is i
                name a) | Equal(v,m) -> Equal(ist v, ist m) | Diff(v,m) ->
                Diff(ist v, ist m) | Derive(m) -> Derive(ist m) | NDerive(m)
                -> NDerive(ist m) | _ -> l
      in match w with | Forall(i,s,a) ->
          let b = expand_quantifiers a ct in let ii = Context.getIndexesOf
          ct s in let tt = map ii (function index -> expand i index s b)
          in if ((len tt) < 2)  then (if tt=[] then True else head tt)
          else fold2 tt (function x -> function y -> And(x,y))
      | Exists(i,s,a) ->
          let b = expand_quantifiers a ct in let ii = Context.getIndexesOf
          ct s in let tt = map ii (function index -> expand i index s b)
          in if ((len tt) < 2)  then (if tt=[] then False else head tt)
          else fold2 tt (function x -> function y -> Or(x,y))
      | And(a,b) -> And(expand_quantifiers a ct, expand_quantifiers b ct) |
      Or(a,b) -> Or(expand_quantifiers a ct, expand_quantifiers b ct) | _ -> w
```

```
let rec listfy =
  function pl ->
    let rec listfyAnd =
      function pa ->
        match pa with | And(a,b) -> (listfyAnd a)@(listfyAnd b) | _ -> [pa]
    in let rec listfyOr =
      function po ->
        match po with | Or(a,b) -> (listfyOr a)@(listfyOr b) | And(_,_) ->
        [listfyAnd po] | _ -> [[po]]
    in listfyOr pl



let rec prnt =
  function p ->
    match p with | Forall(i,n,pl) -> (print_string("{FA "^i^":"^n^"| "); prnt
    pl; print_string("}")) | Exists(i,n,pl) -> (print_string("{EX "^i^":"^n^"|
    "); prnt pl; print_string("}")) | Not(pl) -> (print_string("[not ");
    prnt pl; print_string("]")) | And(pl1,pl2) -> (print_string("{"); prnt
    pl1; print_string(" and "); prnt pl2; print_string("}")) | Or(pl1,pl2)
    -> (print_string("["); prnt pl1; print_string(" or "); prnt pl2;
    print_string("]")) | Equal(v,m) -> (print_string("("); prntMsg v;
    print_string(" = "); prntMsg m; print_string(")")) | Derive(m) ->
    (print_string("( K :> "); prntMsg m; print_string(")")) | Diff(v,m)
    -> (print_string("("); prntMsg v; print_string(" <> "); prntMsg m;
    print_string(")")) | NDerive(m) -> (print_string("( K !:> "); prntMsg
    m; print_string(")")) | True -> (print_string(" T ")) | False ->
    (print_string(" F "))

let rec prntHtml =
  function p ->
    match p with | Forall(i,n,pl) -> (print_string("Forall
    "^n^"<sub>"^i^"</sub>: "); prntHtml pl) | Exists(i,n,pl) ->
    (print_string("Exists "^n^"<sub>"^i^"</sub>: "); prntHtml pl) | Not(pl)
    -> (print_string("not "); prntHtml pl) | And(pl1,pl2) -> (prntHtml pl1;
    print_string(" and "); prntHtml pl2) | Or(pl1,pl2) -> (prntHtml pl1;
    print_string(" or "); prntHtml pl2) | Equal(v,m) -> (prntMsgHtml v;
    print_string(" = "); prntMsgHtml m) | Derive(m) -> (print_string("K :&gt
    "); prntMsgHtml m) | Diff(v,m) -> (prntMsgHtml v; print_string(" &lt &gt
    "); prntMsgHtml m) | NDerive(m) -> (print_string("K !:&gt "); prntMsgHtml
    m) | True -> (print_string(" T ")) | False -> (print_string(" F "))


let prntnplHtml =
  function pln->
    print_list pln ( function ll ->
      print_list ll ( function pll -> prntHtml pll; print_string("<br>")
     ); print_string("<br>")
   )

let prntnpl =
  function pln->
    print_list pln ( function ll ->
```

```
      print_list ll ( function pll -> prnt pll; print_string("\n")
     ); print_string("\n")
   )

let rec normalize =
  function p ->
    function node ->
      let checkequality =
        function a1 ->
          function a2 ->
            let rec compare =
              function l1 ->
                function l2 ->
                  match (l1) with | ll::ls ->
                      if exists l2 (function y -> y=ll) then compare ls l2
                      else false
                  | [] -> true
            in if (len a1) = (len a2) then (
              (compare a1 a2) && (compare a2 a1)
             )
            else false
      in let (ct,asg,k) = Node.getstatus node in let ttt =
      and_or_norm(expand_quantifiers (not_norm p) ct) in let res = remclone
      (listfy ttt) (function x -> function y -> not (checkequality x y)) in res

let rec concretize =
  function pl ->
    function asg ->
      let subst =
        function y ->
          map y (function z ->
            match z with | Equal(v,m) -> Equal(asg v, asg m) | Diff(v,m) ->
            Diff(asg v, asg m) | Derive(m) -> Derive(asg m) | NDerive(m) ->
            NDerive(asg m) | _ -> z
                )
      in map pl (function x -> subst x)


let getatoms = function n1 -> function n2 -> let ttt = couplify n1 n2 in
let res = map ttt (
                   function (a,b) ->
let trr= couplify a b in let atoms = filter trr (function (a1, a2) ->
a2=False && (a1<>False)) in map atoms (fst) ) in remclone (debox res) (<>)
```

Message

```
type pname = string*int type variable = string*pname type svtype =
  | Gen | GenR | SKey | Name | RName | PMKey | MPKey
```

```
type msg =
  | K of string*int | PN of string*int | PNp of string*int | PNm of string*int
  | NO of string*int

  | Var of variable | BVar of variable | SVar of variable*int*svtype

  | Crypt of msg*msg | Couple of msg*msg | Publ of msg | Priv of msg |
  Bottom | Trick of (msg*msg) list


let setVariable ((name,index),(namevar,(namep, indexp))) =
(namevar,(name,index))


let notSVar =
  function m ->
    match m with | SVar(_,_,_) -> false | _ -> true

let notVar =
  function m ->
    match m with | Var(_) -> false | _ -> true

let compl =
  function kk ->
    match kk with | K(_) -> kk | PNp(s,p) -> PNm(s,p) | PNm(s,p) -> PNp(s,p)
    | _ -> Bottom

let toName =
  function
    | PNp(p,i) -> PN(p,i) | PNm(p,i) -> PN(p,i) | _ -> Bottom

let toPNp =
  function
    | PN(p,i) -> PNp(p,i) | _ -> Bottom

let toPNm =
  function
    | PN(p,i) -> PNm(p,i) | _ -> Bottom

let isKey =
  function k ->
    match k with | K(_) | PNp(_,_) | PNm(_,_) | Var(_) | SVar(_,_,_) |
    Publ(Var(_)) | Priv(Var(_)) | Publ(SVar(_)) | Priv(SVar(_)) -> true |
    _ -> false

let isGroundKey =
  function k ->
    match k with | K(_) | PNp(_,_) | PNm(_,_) -> true | _ -> false

let rec checkmsg =
  function m ->
    match m with | Crypt(m1,k) -> (checkmsg m1)&&isKey(k) | Couple(m1,m2) ->
    (checkmsg m1)&&(checkmsg m2) | Publ(K(_)) | Publ(PNp(_)) | Publ(PNm(_))
    | Priv(K(_)) | Priv(PNp(_)) | Priv(PNm(_)) -> false | _ -> true
```

```
let rec isBinding =
  function (m,v) ->
    match m with | Couple(m1,m2) -> isBinding(m1,v)||isBinding(m2,v) |
    Crypt(m1,_) -> isBinding(m1,v) | BVar(var) -> Var(var)=v | _ -> false

let prntVar =
  function (s,(a,i)) -> print_string(s^"_"^a^"_"^string_of_int(i))

let prntVarHtml =
  function (s,(a,i)) -> print_string(s^"<sub>"^string_of_int(i)^"</sub>")

let tostringVar =
  function (s,(a,i)) -> (s^"_"^a^"_"^string_of_int(i))


let rec prntVarlist =
  function vs -> match vs with | x::y::xs -> prntVar x; print_string(",
  "); prntVarlist (y::xs) | x::[] -> prntVar x | [] -> ()

let rec prntVarlistHtml =
  function vs -> match vs with | x::y::xs -> prntVarHtml x; print_string(",
  "); prntVarlistHtml (y::xs) | x::[] -> prntVarHtml x | [] -> ()

let rec tostringVarList =
  function vs -> match vs with | x::y::xs -> (tostringVar x)^",
  "^(tostringVarList (y::xs)) | x::[] -> tostringVar x | [] -> ""


let rec prntMsgHtml =
  function s ->
    match s with | (K(m,i)) ->
    print_string(m^"<sub>"^string_of_int(i)^"</sub>") | (PN(s,i)) ->
    print_string(s^"<sub>"^string_of_int(i)^"</sub>") | (PNp(s,i)) ->
    print_string(s^"<sub>"^string_of_int(i)^"</sub>"^"<sup>+</sup>")
    | (PNm(s,i)) ->
    print_string(s^"<sub>"^string_of_int(i)^"</sub>"^"<sup>-</sup>") |
    (NO(s,i)) -> print_string (s^"<sub>"^string_of_int(i)^"</sub>") |
    Crypt(m1,m2) -> (print_string "{";prntMsgHtml m1; print_string
    "}"; print_string "<sub>"; prntMsgHtml m2; print_string
    "</sub>") | Couple(m1,m2) -> (prntMsgHtml m1; print_string ",";
    prntMsgHtml m2;) | (Var(s,(a,i))) | (BVar(s,(a,i))) -> prntVarHtml
    (s,(a,i)) | (SVar((s,(a,i)),n, Gen)) -> ((prntVarHtml (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":*)")) | (SVar((s,(a,i)),n, GenR)) ->
    ((prntVarHtml (s,(a,i))); print_string("(K:"^string_of_int(n)^":*r)"))
    | (SVar((s,(a,i)),n, SKey)) -> ((prntVarHtml (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":sk)")) | (SVar((s,(a,i)),n, Name)) ->
    ((prntVarHtml (s,(a,i))); print_string("(K:"^string_of_int(n)^":n"))
    | (SVar((s,(a,i)),n, RName)) -> ((prntVarHtml (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":nr)")) | (SVar((s,(a,i)),n, MPKey))
    -> ((prntVarHtml (s,(a,i))); print_string("(K:"^string_of_int(n)^":mp)"))
    | (SVar((s,(a,i)),n, PMKey)) -> ((prntVarHtml (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":pm)")) | Publ(Var(v)) ->
```

```
    (prntVarHtml v; print_string("<sup>+</sup>")) | Priv(Var(v)) ->
    (prntVarHtml v; print_string("<sup>-</sup>")) | Publ(SVar(v,i,t))
    -> (prntMsgHtml(SVar(v,i,t)); print_string("<sup>+</sup>"))
    | Priv(SVar(v,i,t)) -> (prntMsgHtml(SVar(v,i,t));
    print_string("<sup>-</sup>")) | Publ(nn) -> (prntMsgHtml(nn);
    print_string("<sup>+</sup>")) | Priv(nn) -> (prntMsgHtml(nn);
    print_string("<sup>-</sup>")) | Bottom -> print_string(".")  | Trick(_)
    -> print_string("Trick")

let rec prntMsg =
  function s ->
    match s with | (K(m,i)) -> print_string(m^"_"^string_of_int(i)) |
    (PN(s,i)) -> print_string(s^"_"^string_of_int(i)) | (PNp(s,i))
    -> print_string(s^"_"^string_of_int(i)^"+") | (PNm(s,i)) ->
    print_string(s^"_"^string_of_int(i)^"-") | (NO(s,i)) -> print_string
    (s^"_"^string_of_int(i)) | Crypt(m1,m2) -> (print_string "{";prntMsg
    m1; print_string "}"; prntMsg m2) | Couple(m1,m2) -> (prntMsg m1;
    print_string ","; prntMsg m2;) | (Var(s,(a,i))) | (BVar(s,(a,i))) ->
    prntVar (s,(a,i)) | (SVar((s,(a,i)),n, Gen)) -> ((prntVar (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":gen)")) | (SVar((s,(a,i)),n, GenR))
    -> ((prntVar (s,(a,i))); print_string("(KR:"^string_of_int(n)^":gen)"))
    | (SVar((s,(a,i)),n, SKey)) -> ((prntVar (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":skey)")) | (SVar((s,(a,i)),n, Name))
    -> ((prntVar (s,(a,i))); print_string("(K:"^string_of_int(n)^":name)"))
    | (SVar((s,(a,i)),n, RName)) -> ((prntVar (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":rname)")) |
    (SVar((s,(a,i)),n, MPKey)) -> ((prntVar (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":mpkey)")) |
    (SVar((s,(a,i)),n, PMKey)) -> ((prntVar (s,(a,i)));
    print_string("(K:"^string_of_int(n)^":pmkey)")) | Publ(Var(v))
    -> (prntVar v; print_string("+")) | Priv(Var(v)) -> (prntVar v;
    print_string("-")) | Publ(SVar(v,i,t)) -> (prntMsg(SVar(v,i,t));
    print_string("+")) | Priv(SVar(v,i,t)) -> (prntMsg(SVar(v,i,t));
    print_string("-")) | Publ(nn) -> (prntMsg(nn); print_string("+")) |
    Priv(nn) -> (prntMsg(nn); print_string("-")) | Bottom -> print_string(".")
    | Trick(_) -> print_string("Trick")

let rec tostring =
  function s ->
    match s with | (K(m,i)) -> (m^"_"^string_of_int(i)) | (PN(s,i))
    ->(s^"_"^string_of_int(i)) | (PNp(s,i)) -> (s^"_"^string_of_int(i)^"+")
    | (PNm(s,i)) -> (s^"_"^string_of_int(i)^"-") | (NO(s,i)) ->
    (s^"_"^string_of_int(i)) | Crypt(m1,m2) -> "{"^(tostring m1)^"}"^(tostring
    m2) | Couple(m1,m2) -> ((tostring m1)^","^(tostring m2)) | (Var(s,(a,i)))
    | (BVar(s,(a,i))) -> tostringVar (s,(a,i)) | (SVar((s,(a,i)),n,Gen))
    -> ((tostringVar (s,(a,i)))^("(KG:"^string_of_int(n)^")"))
    | (SVar((s,(a,i)),n,GenR)) -> ((tostringVar
    (s,(a,i)))^("(KRG:"^string_of_int(n)^")")) | (SVar((s,(a,i)),n,SKey))
    -> ((tostringVar (s,(a,i)))^("(KSK:"^string_of_int(n)^")"))
    | (SVar((s,(a,i)),n,Name)) -> ((tostringVar
    (s,(a,i)))^("(KN:"^string_of_int(n)^")")) | (SVar((s,(a,i)),n,RName))
    -> ((tostringVar (s,(a,i)))^("(KRN:"^string_of_int(n)^")"))
    | (SVar((s,(a,i)),n,PMKey)) -> ((tostringVar
    (s,(a,i)))^("(KPM:"^string_of_int(n)^")")) | (SVar((s,(a,i)),n,MPKey)) ->
```

```
    ((tostringVar (s,(a,i)))^("(KMP:"^string_of_int(n)^")")")) | Publ(Var(v))
    -> (tostringVar v)^("+") | Priv(Var(v)) -> (tostringVar v)^("-") |
    Publ(SVar(v,i,t)) -> (tostring(SVar(v,i,t))^"+") | Priv(SVar(v,i,t)) ->
    (tostring(SVar(v,i,t))^"-") | Publ(nn) -> (tostring(nn)^"+") | Priv(nn) ->
    (tostring(nn)^"-") | Bottom -> (".")  | Trick(_) -> ("Trick")

let rec hasntFreeVar =
  function m ->
    let rec checkall =
      function mm ->
        match mm with | Couple(m1,m2) -> (checkall m1) && checkall(m2)
        | Crypt(m1, kk) when isGroundKey kk-> (checkall m1) | Crypt(m1,
        Var(v)) | Crypt(m1, Publ(Var(v))) | Crypt(m1, Priv(Var(v))) ->
        (checkall m1)&&isBinding(m,Var(v)) | Var(_) -> false | Publ(Var(_))
        -> false | Priv(Var(_)) -> false | _ -> true
    in checkall m

let ispkey =
  function m ->
    match m with | PNp(_,_) -> true | _ -> false

let ismkey =
  function m ->
    match m with | PNm(_,_) -> true | _ -> false

let isskey =
  function m ->
    match m with | K(_) -> true | _ -> false

let isname =
  function m ->
    match m with | PN(_,_) -> true | _ -> false

let iscrypt =
  function m ->
    match m with | Crypt(_,_) -> true | _ -> false



let mintype =
  function (t1,t2) ->
    let tval =
      function t ->(
        match t with | Gen -> 4 | GenR -> 3 | Name -> 1 | SKey -> 1 | RName ->
        0 | _ -> -1)
    in if tval(t1)>tval(t2) then t2 else t1

let comparable =
  function (t1,t2) ->
    ((t1<>Name && t1<>RName)||(t2<>SKey))&& ((t2<>Name &&
    t2<>RName)||(t1<>SKey))

let rec typeOfVar =
  function m ->
```

```
    function v ->
      match m with | Couple(m1,m2) ->
          mintype(typeOfVar m1 v, typeOfVar m2 v)
      | Crypt(m1, Var(vv)) when v=vv -> SKey | Publ(Var(vv)) when v=vv -> Name
      | Priv(Var(vv)) when v=vv -> Name | Crypt(m1, Publ(Var(vv))) when vv=v
      -> Name | Crypt(m1, Priv(Var(vv))) when vv=v -> Name | Crypt(m1, kk) ->
          mintype(typeOfVar m1 v, typeOfVar kk v)
      | _ -> Gen
```

Node

```
open Status open Principal open Message open Assignment open Process open
Action

type redex = principal*action*principal

type noderecord = {
    mutable status: status; mutable numprinc: int; mutable numjoin: int;
    mutable father: node; mutable label: action; mutable redextable: redex
    list; mutable jointable: action list; mutable closed: bool; mutable
    inorout: bool; mutable joining: bool; mutable nsons: int; mutable id:
    int; mutable level: int

} and node = NulNode | Node of noderecord

let kid = ref(0)

let create =
  function st -> kid:=!kid+1;
    let nn = {
      status = st; numprinc = 0; numjoin = 0; father = NulNode; label =
      NoMoreAct; redextable = []; jointable = []; closed = false; inorout =
      true; joining = true; nsons = 0; id = !kid; level = 0;
    } in Node(nn)


let setstatus =
  function Node(nn),stl -> nn.status <- stl

let setlabel =
  function Node(nn),lbl -> nn.label <- lbl

let settable =
  function Node(nn),rll -> nn.redextable <- rll

let setjoins =
  function Node(nn),jjr -> nn.jointable <- jjr

let addson =
  function Node(nn) ->
```

```
    nn.nsons <- (nn.nsons+1)



let setnd =
  function Node(nn),npp -> nn.numprinc <- npp

let setnj =
  function Node(nn),njj -> nn.numjoin <- njj

let close =
  function Node(nn) -> nn.closed <- true

let setIn =
  function Node(nn) -> nn.inorout <- false

let setJn =
  function Node(nn) -> nn.joining <- false

let getlevel =
  function Node(nn) -> (nn.level)

let setfather =
  function Node(nn),fth -> (
    nn.father <- fth; nn.level <- ((getlevel fth)+1); addson fth
   )



let getlabel =
  function Node(nn) -> (nn.label)

let getfather =
  function Node(nn) -> (nn.father)

let getnd =
  function Node(nn) -> (nn.numprinc)

let getnj =
  function Node(nn) -> (nn.numjoin)

let getstatus =
  function Node(nn) -> (nn.status)

let gettable =
  function Node(nn) -> (nn.redextable)

let getjoins =
  function Node(nn) -> (nn.jointable)

let getSons =
  function Node(nn) -> (nn.nsons)
```

```
let getRedex =
  function Node(nn) ->
    match (nn.redextable) with | x::xs -> (nn.redextable <- xs; x) |
    [] when (nn.closed) -> (("",0,[],Nil),Closed,("",0,[],Nil)) | [] ->
    (("",0,[],Nil),NoMoreAct,("",0,[],Nil))

let getJoin =
  function Node(nn) ->
    match (nn.jointable) with | x::xs -> (nn.jointable <- xs; x) | [] when
    nn.closed -> (Closed) | [] -> (NoMoreAct)


let getid =
  function Node(nn) -> nn.id
    | _ -> -1

let isNul =
  function NulNode -> true
    | Node (_) -> false

let isOut =
  function NulNode -> true
    | Node (nn) -> nn.inorout

let isJn =
  function NulNode -> true
    | Node (nn) -> nn.joining

let isLeaf =
  function NulNode -> false
    | Node (nn) -> (nn.nsons = 0)


let nulNode =
  function () -> NulNode


let isTerminal =
  function Node(nn) -> Status.isTerminal (nn.status)

let isPartiallyTerminated =
  function Node(nn) -> Status.isPartiallyTerminated (nn.status)




let rec prntT =
  function ll ->
    match ll with | [] -> print_string("\n"); | (p1,Input(pp,m,s),p2)::ls -> (
        print_string("Input <"); prntMsg (s m); print_string(">\n"); prntT ls
       )
    | (p1,Output(pp,m,s),p2)::ls -> (
        print_string("Output <"); prntMsg (s m); print_string(">\n"); prntT ls
       )
```

```
let rec prntJ =
  function ll ->
    match ll with | [] -> print_string("\n"); | (Join(p,s))::ls -> (
        print_string("Join <"); print_string(fst(p)^"_"^string_of_int(snd(p)));
        print_string(">\n"); prntJ ls
       )

let prnt =
  function Node(nn) ->
    (
     print_string("\n--------------------------------\nAction: ");
     Action.prnt (nn.label); print_string("\n   ||\n   ||\n   \\/\n");
     print_string("\n-----\nNode: id="^string_of_int(nn.id)^"
     lev="^string_of_int(nn.level)^"\n-----\nSon
     of:"^string_of_int(getid(nn.father))^"\n"); Status.prnt (nn.status)
    ) | NulNode -> print_string("\nNode: NULNODE\n")
```

Parsecmdline

```
type graphopt = {
        mutable context: bool; mutable actions: bool; mutable k: bool;
        mutable asg: bool; mutable level: int;
}

type options = {
    mutable output_type: int; mutable attack_type: int; mutable pr_file:
    string; mutable pl_file: string; mutable pj_file: string; mutable kb_file:
    string; mutable trace_terminal: bool; mutable go: graphopt; mutable errmsg:
    string; mutable help: bool; mutable max_princ: int; mutable interactive:
    bool; mutable feedback: bool; mutable lazyness: bool; mutable mga:bool;
    mutable ic: int list; mutable greed: bool;
  }


let init_options =
  function () ->
    let t = { output_type = -1;
              attack_type = 0; pr_file = ""; pl_file = ""; pj_file = "";
              kb_file = ""; trace_terminal = false; go = { actions = false;
              k = false; asg = false; context = false; level = 16384;
            };
              errmsg = ""; help = false; max_princ = 0; interactive =
              false; feedback = false; lazyness = true; mga = false; ic =
              []; greed = false;
            } in t


let initgo =
  function n ->
    function lev -> {
     context = (n >= 8); actions = (n mod 8 >= 4); k = (n mod 4 >= 2 ); asg =
```

```
    (n mod 2 = 1); level = lev;
  }


let parsecmdline =
  function argv ->
    function opts ->
      let i = ref(1) in let n = Array.length argv in let rec parse =
        function () ->
          if !i<n then (
            match argv.(!i) with | "-T" -> (opts.output_type <- 0; i:= !i+1;
            parse()) | "-G" ->
                opts.output_type <- 1; if (!i+1<n) then (
                  try (
                    let mm = int_of_string(argv.(!i+1)) in (opts.go <-
                    initgo mm opts.go.level; i:=!i+2; parse())
                   ) with _ -> (opts.errmsg <- "-G requires numeric option")
                 )
                else (opts.errmsg <- "-G requires numeric option")
            | "-l" ->
                if (!i+1<n) then (
                  try (
                    let mm = int_of_string(argv.(!i+1)) in (opts.go.level <-
                    mm; i:=!i+2; parse())
                   ) with _ -> (opts.errmsg <- "-l requires numeric option")
                 )
                else (opts.errmsg <- "-G requires numeric option")
            | "-V" -> (opts.output_type <- 2; i:= !i+1; parse()) |
            "-L" -> (opts.attack_type <- 1; i:= !i+1; parse()) | "-H" ->
            (opts.attack_type <- 2; i:= !i+1; parse()) | "-h" | "--help" ->
            (opts.help <- true; i:= !i+1; parse()) | "-lazy" -> (opts.lazyness
            <- false; i:= !i+1; parse()) | "-mga" -> (opts.mga <- true; i:=
            !i+1; parse()) | "-t" | "--terminal" -> (opts.trace_terminal
            <- true; i:= !i+1; parse()) | "--interactive" | "-i"  ->
            (opts.interactive <- true; i:= !i+1; parse()) | "--feedback" |
            "-f"  -> (opts.feedback <- true; i:= !i+1; parse()) | "-g" |
            "--greed" ->(opts.greed <- true; i:= !i+1; parse())

            | "-s" | "--session"->
                if (!i+1<n) then (
                  let rec roles =
                    function () ->
                      try (
                        let mm = int_of_string(argv.(!i+1)) in (opts.ic <-
                        mm::(opts.ic); i:=!i+1); roles();
                       ) with _ -> i:=!i+1
                  in roles(); parse(); )
                else (opts.errmsg <- "number of roles in a session required")



            | "-max" | "-nprinc" ->
                if (!i+1<n) then (
                  try (
```

```
                    let mm = int_of_string(argv.(!i+1)) in (opts.max_princ <-
                    mm ; i:=!i+2; parse())
                   ) with _ -> (opts.errmsg <- "number of principal in a
                   session required")
                 )
                else (opts.errmsg <- "number of principal in a session
                required")
            | s ->
                if (opts.pr_file = "") then  (opts.pr_file <- s; i:=!i+1;
                parse()) else
                  if (opts.pl_file = "") then  (opts.pl_file <- s; i:=!i+1;
                  parse()) else
                    if (opts.pj_file = "") then  (opts.pj_file <- s; i:=!i+1;
                    parse()) else
                      if (opts.kb_file = "") then  (opts.kb_file <- s;
                      i:=!i+1; parse()) else (opts.errmsg <- "don't know
                      what to do with "^s)

           ) else ()
      in parse()




let synopsis =
  function flag ->
    print_string("\nUsage: tool principals logic join knowledge  [-GVT]
    [-max nprinc] [-if]\n\n"); if flag then (
      print_string("\n   -T: Output is text representing the set of traces");
      print_string("\n   -G param: Output is a GraphViz compliant file
      representing state space tree"); print_string("\n      param: a numeric
      value [0..15] specifing what to print. See manual"); print_string("\n
      -V: Enable verification of formula. Output is the description of
      attacks");

      print_string("\n   -t, --terminal: with -T prints out terminal
      traces only"); print_string("\n   -i, --interactive: with -V asks for
      continuation after the discovery of an attack"); print_string("\n   -f,
      --feedback: prints on stderr info on search status");

      print_string("\n   -h, --help: print this help");

      print_string("\n")
     ) else ()


let htmlinit =
  function () ->
    print_string("<html><body>")
let htmldone =
  function () ->
    print_string("</body></html>")
```

Parser

```
open Tokenizer open Message open Principal open Process

type pars =
  | Pexp of process | Dot | Plp | Ino of msg | Outo of msg | Sumo | Paro | End


exception ParseError exception CantReduce

let errOK                                        = 0;;
let errCOMMENT_NOT_OPENED                        = 1;; let
errCOMMENT_NOT_CLOSED                        = 2;; let errUNKNOWN_TOKEN
= 3;; let errMISSING_OPENING_ROUND_PAR               =
4;; let errMISSING_CLOSING_ROUND_PAR               =
5;; let errMISSING_OPENING_SQUARE_PAR              =
6;; let errMISSING_CLOSING_SQUARE_PAR              = 7;;
let errMISSING_OPENING_CURLY_PAR             = 8;; let
errMISSING_CLOSING_CURLY_PAR                 = 9;; let errMISSING_COMMA
= 10;; let errMISSING_SEMICOLON                      = 11;; let
errMISSING_COLON                             = 12;; let errEXPECTED_IDE
= 13;; let errUNEXPECTED_TOKEN                       = 14;; let errWRONG_KEY
= 15;; let errWRONG_VAR                              = 16;; let
errMALFORMED_EXPR                            = 17;; let errUPPERCASE_PRINCIPAL
= 18;; let errBINDING_VARIABLE_AS_A_KEY              = 19;;


let errorString = Array.create 20 "";; errorString.(errOK) <-
"Ok.";; errorString.(errCOMMENT_NOT_OPENED) <- "Comment not
opened";; errorString.(errCOMMENT_NOT_CLOSED) <- "Comment
not closed";; errorString.(errUNKNOWN_TOKEN) <- "Unknown
token";; errorString.(errMISSING_OPENING_ROUND_PAR) <-
"Missing (";; errorString.(errMISSING_CLOSING_ROUND_PAR) <-
"Missing )";; errorString.(errMISSING_OPENING_SQUARE_PAR) <-
"Missing [";; errorString.(errMISSING_CLOSING_SQUARE_PAR) <-
"Missing ]";; errorString.(errMISSING_OPENING_CURLY_PAR) <-
"Missing {";; errorString.(errMISSING_CLOSING_CURLY_PAR) <-
"Missing }";; errorString.(errMISSING_COMMA) <- "Missing
,";; errorString.(errMISSING_SEMICOLON) <- "Missing ;";;
errorString.(errMISSING_COLON) <- "Missing :";; errorString.(errEXPECTED_IDE)
<- "Identifier expected";; errorString.(errUNEXPECTED_TOKEN) <- "Unexpected
token";; errorString.(errWRONG_KEY) <- "Wrong key";; errorString.(errWRONG_VAR)
<- "Wrong variable";; errorString.(errMALFORMED_EXPR) <- "Malformed
expression";; errorString.(errUPPERCASE_PRINCIPAL) <- "Principal name should
be uppercase";; errorString.(errBINDING_VARIABLE_AS_A_KEY) <- "A binding
variable cannot occur as a key";;


let whereError =
  function () -> "
  ("^(string_of_int(fst(where())))^","^(string_of_int(snd(where())))^")"
```

```
let errorCode = ref(0) let errorData = ref ("") let errorMsg =
  function () -> "ER"^string_of_int(!errorCode)^": "^errorString.(!errorCode)^"
  "^(!errorData)^" "^whereError()^"\n"


let putError =
        function (code, s) -> errorData:= s; errorCode:=code

let getErrorMsg = function () -> errorMsg() let getErrorCode = function ()
-> !errorCode


let isUpper =
  function s ->
    let isUp =
      function ch ->
        match ch with | 'A'..'Z' -> true | _ ->  false
    in let rec isU =
      function (s, i, m) ->
        if (i<m) then
          if isUp(s.[i]) then isU(s, i+1, m) else false
        else true
    in isU(s,0, String.length s)


let rec ignoreComment =
  function () ->
    let a = getToken() in match a with | LSym("*") ->
        let b = getToken() in if b=LSym("/") then () else (pushback(b);
        ignoreComment())
    | LStop -> (raise ParseError) | _ -> ignoreComment()


let rec readToken =
  function () ->
    let a = getToken() in match a with | LSym("/") ->
        let b = getToken() in if b=LSym("/") then (flushline(); readToken())
        else
          if b=LSym("*") then ((let ss = whereError() in try (ignoreComment())
          with ParseError -> putError(errCOMMENT_NOT_CLOSED,ss)); readToken())
          else (pushback(b); a)
    | LSym("*") ->
        let b = getToken() in if b=LSym("/") then
        (putError(errCOMMENT_NOT_OPENED,""); raise ParseError) else
        (pushback(b); a)
    | LUnknown(a) -> (putError(errUNKNOWN_TOKEN,""); raise ParseError) | _ -> a


let rec getMsg =
  function () ->
    let p1 = readToken() in if p1<>LSym("(") then
    (putError(errMISSING_OPENING_ROUND_PAR,""); raise ParseError) else
      let ms = getMessage() in let p2 = readToken() in if p2<>LSym(")") then
      ( putError(errMISSING_CLOSING_ROUND_PAR,""); raise ParseError) else ms
```

```
and getMessage =
  function() ->
    let a = readToken() in match a with | LSym("(") -> getCouple() | LSym("{")
    -> getCrypt() | LSym("?") -> (
        let y=getVal() in match y with | (Var(s,(a,b))) -> (BVar(s,(a,b))) |
        Publ(v) | Priv(v) -> (putError(errBINDING_VARIABLE_AS_A_KEY,""); raise
        ParseError) | _ -> (putError(errUNKNOWN_TOKEN,""); raise ParseError)
       )
    | LIde(v) -> pushback(a); getVal() | _ -> (putError(errUNKNOWN_TOKEN,"");
    raise ParseError)
and getCouple =
  function () ->
    let m1 = getMessage() in let v1 = readToken() in if v1<>LSym(",") then
    (putError(errMISSING_COMMA,""); raise ParseError) else
      let m2 = getMessage() in let v2 = readToken() in if v2<>LSym(")")
      then (putError(errMISSING_CLOSING_ROUND_PAR,""); raise ParseError)
      else Couple(m1,m2)
and getCrypt =
  function () ->
    let m1 = getMessage() in let v1 = readToken() in if v1<>LSym("}") then
    (putError(errMISSING_CLOSING_CURLY_PAR,""); raise ParseError) else
      let k1 = getVal() in Crypt(m1,k1)
and getVal =
  function () ->
    let a = readToken() in match a with | LIde(s) when isUpper(s) -> (
        let k = readToken() in match k with | LSym("+") -> (PNp(s,0)) |
        LSym("-") -> (PNm(s,0)) | LSym(c) -> pushback(k); (PN(s,0)) | _
        ->(putError(errWRONG_KEY,""); raise ParseError)
       )
    | LIde(s) -> (
        if s.[0] = 'n' then (NO(s,0)) else
          if s.[0] = 'k' then (K(s,0)) else (
            let k = readToken() in match k with | LSym("+") ->
            Publ(Var(s,("",0))) | LSym("-") -> Priv(Var(s,("",0))) | LSym(c)
            -> pushback(k); (Var(s,("",0))) | _ ->(putError(errWRONG_VAR,"");
            raise ParseError)
           )
       )
    | _ -> (putError(errEXPECTED_IDE,""); raise ParseError);;



let reducer =
  function (stack, pr) ->
    match stack with | Ino(m)::st -> Pexp(In(m,Nil))::st |
    Outo(m)::st -> Pexp(Out(m,Nil))::st | Pexp(e)::Dot::Outo(m1)::st ->
    Pexp(Out(m1,e))::st | Pexp(e)::Dot::Ino(m1)::st -> Pexp(In(m1,e))::st
    | Pexp(e1)::Sumo::Pexp(e2)::st -> Pexp(Sum(e2,e1))::st |
    Pexp(e1)::Paro::Pexp(e2)::st when pr<>Sumo -> Pexp(Par(e2,e1))::st | _
    -> raise CantReduce;;


let rec parseExp =
  function (tok, stack) ->
```

```
    match (tok, stack) with | LIde("in"), _ -> let ms = getMsg() in
    Ino(ms)::stack | LIde("out"), _ -> let ms = getMsg() in Outo(ms)::stack
    | LSym("."), _ -> Dot::stack | LSym("+"), _ -> (try parseExp (tok,
    reducer(stack, Sumo)) with CantReduce -> Sumo::stack) | LSym("|"), _ ->
    (try parseExp (tok, reducer(stack, Paro)) with CantReduce -> Paro::stack)
    | LSym("("), _ -> Plp::stack | LSym(")"), Pexp(e)::Plp::st -> Pexp(e)::st
    | LSym(")"), _ -> parseExp (tok, reducer(stack, Dot)) | LSym("]"), _ ->
    End::stack | _ -> putError(errUNKNOWN_TOKEN,""); raise ParseError;;


let rec reduce_all =
        function stack ->
                match stack with
                        | [Pexp(e)] -> e | [] ->
                        (putError(errMISSING_SEMICOLON,"");Nil) | _ ->
                        reduce_all ( reducer (stack, Dot));;


let rec parser =
    function stack ->
      let a = readToken() in (match a with | LStop -> [] | _ ->
          let ss = parseExp (a, stack) in let c = List.hd ss in if c=End
          then stack else parser ss
      );;


let parseName =
  function () ->
    let a = readToken() in match a with | LIde(n) when isUpper(n) -> (
        let s = readToken() in if s<>LSym(":") then
        (putError(errMISSING_COLON,""); raise ParseError) else n
       )
    | _ -> (putError(errUPPERCASE_PRINCIPAL,""); raise ParseError);;

let rec parseVariables =
  function () ->
    let p1 = readToken() in if p1<>LSym("(") then
    (putError(errMISSING_OPENING_ROUND_PAR,""); raise ParseError) else
      let vs = parseVars([]) in let p2 = readToken() in if p2<>LSym(")") then
      (putError(errMISSING_CLOSING_ROUND_PAR,""); raise ParseError) else vs
and parseVars =
  function ll ->
    let a = readToken() in match (a,ll) with | (LSym(")"), _) -> pushback(a);
    ll | (LIde(s), _ ) -> (
        let a2 = readToken() in if (a2<> LSym(",") && a2<>LSym(")") )
        then (putError(errUNEXPECTED_TOKEN,""); raise ParseError) else
        (pushback(a2); parseVars(ll@[(s,("",0))]))
       )
    | (LSym(","), l::ls) -> (
        let a2 = readToken() in match a2 with | LIde(j) -> (pushback(a2);
        parseVars(ll)) | _ -> (putError(errUNEXPECTED_TOKEN,""); raise
        ParseError)
       )
    | _ -> (putError(errUNEXPECTED_TOKEN,""); raise ParseError);;
```

```
let parsePrincipal =
  function () ->
    let (fl1, name) = (try ((true,parseName())) with ParseError -> (false,""))
    in if not fl1 then Principal.create("",-1, [], Nil) else
      let (fl2,vars) = (try ((true,parseVariables())) with ParseError ->
      (false,[])) in if not fl2 then Principal.create("",-1, [], Nil) else
        let p1 = readToken() in if p1<>LSym("[") then
        (putError(errMISSING_OPENING_SQUARE_PAR,""); Principal.create("",-1,
        [], Nil)) else
          let (fl3,ex) = (
            try ((true, reduce_all (parser []))) with | CantReduce ->
            (putError(errMALFORMED_EXPR,""); (false,Nil)) | ParseError ->
            (false, Nil)
           )
          in if fl3 then
            let p2 = readToken() in if p2<>LSym(";") then
            (putError(errMISSING_SEMICOLON,""); Principal.create("",-1,
            [], Nil)) else (putError(errOK,""); Principal.create(name, 0,
            vars, ex))
          else Principal.create("",-1, [], Nil)


let more_parsing =
        function () ->
                let a = readToken() in (pushback(a); a <> LStop)

let init_parser =
        function fname -> init_tokenizer(fname);;
let done_parser =
        function () -> done_tokenizer();;
```

Pl_parser

```
open Tokenizer open Message open Logic

type pars =
        | Pexp of pls | PNot | PAnd | POr | Impl | DImpl | Fa of string*string
        | Ex of string*string | Plp | Token of string | End


exception ParseError exception CantReduce


let errOK                                       = 0;;
let errCOMMENT_NOT_OPENED                       = 1;; let
errCOMMENT_NOT_CLOSED                   = 2;; let errUNKNOWN_TOKEN
= 3;; let errMISSING_OPENING_ROUND_PAR                =
4;; let errMISSING_CLOSING_ROUND_PAR                = 5;; let
```

```
errMISSING_SEMICOLON                        = 6;; let errMISSING_COLON
= 7;; let errMISSING_DOT                              = 8;; let
errEXPECTED_IDE                             = 9;; let errUNEXPECTED_TOKEN
= 10;; let errWRONG_KEY                               = 11;; let
errWRONG_VAR                                = 12;; let errMALFORMED_EXPR
= 13;; let errUPPERCASE_PRINCIPAL                     = 14;; let
errMISSING_VARIABLE                         = 15;; let errMISSING_COMMA
= 16;; let errMISSING_OPENING_CURLY_PAR               = 17;; let
errMISSING_CLOSING_CURLY_PAR                = 18;; let errMISSING_INDEX
= 19;;



let errorString = Array.create 20 "";; errorString.(errOK) <-
"Ok.";; errorString.(errCOMMENT_NOT_OPENED) <- "Comment not
opened";; errorString.(errCOMMENT_NOT_CLOSED) <- "Comment
not closed";; errorString.(errUNKNOWN_TOKEN) <- "Unknown
token";; errorString.(errMISSING_OPENING_ROUND_PAR) <-
"Missing (";; errorString.(errMISSING_CLOSING_ROUND_PAR) <-
"Missing )";; errorString.(errMISSING_OPENING_CURLY_PAR) <-
"Missing {";; errorString.(errMISSING_CLOSING_CURLY_PAR) <-
"Missing }";; errorString.(errMISSING_SEMICOLON) <- "Missing ;";;
errorString.(errMISSING_COLON) <- "Missing :";; errorString.(errMISSING_DOT)
<- "Missing .";; errorString.(errMISSING_COMMA) <- "Missing
,";; errorString.(errEXPECTED_IDE) <- "Identifier expected";;
errorString.(errUNEXPECTED_TOKEN) <- "Unexpected token";;
errorString.(errWRONG_KEY) <- "Wrong key";; errorString.(errWRONG_VAR) <-
"Wrong variable";; errorString.(errMALFORMED_EXPR) <- "Malformed expression";;
errorString.(errUPPERCASE_PRINCIPAL) <- "Principal name should be uppercase";;
errorString.(errMISSING_VARIABLE) <- "Missing variable in equality";;
errorString.(errMISSING_INDEX) <- "Missing index";;



let whereError =
  function () -> "
  ("^(string_of_int(fst(where())))^","^(string_of_int(snd(where())))^")"


let errorCode = ref(0) let errorData = ref ("") let errorMsg =
  function () -> "ER"^string_of_int(!errorCode)^": "^errorString.(!errorCode)^"
  "^(!errorData)^" "^whereError()^"\n"


let putError =
  function (code, s) -> errorData:= s; errorCode:=code

let getErrorMsg = function () -> errorMsg() let getErrorCode = function ()
-> !errorCode


let isUpper =
  function s ->
    let isUp =
```

```
      function ch ->
        match ch with | 'A'..'Z' -> true | '_' -> true | _ ->  false
    in let rec isU =
      function (s, i, m) ->
        if (i<m) then
          if isUp(s.[i]) then isU(s, i+1, m) else false
        else true
    in isU(s,0, String.length s)


let rec ignoreComment =
  function () ->
    let a = getToken() in match a with | LSym("*") ->
        let b = getToken() in if b=LSym("/") then () else (pushback(b);
        ignoreComment())
    | LStop -> (raise ParseError) | _ -> ignoreComment()


let rec readToken =
  function () ->
    let a = getToken() in match a with | LSym("/") ->
        let b = getToken() in if b=LSym("/") then (flushline(); readToken())
        else
          if b=LSym("*") then ((let ss = whereError() in try (ignoreComment())
          with ParseError -> putError(errCOMMENT_NOT_CLOSED,ss)); readToken())
          else (pushback(b); a)
    | LSym("*") ->
        let b = getToken() in if b=LSym("/") then
        (putError(errCOMMENT_NOT_OPENED,""); raise ParseError) else
        (pushback(b); a)
    | LUnknown(a) -> (putError(errUNKNOWN_TOKEN,""); raise ParseError) | _ -> a

let getName =
  function () ->
    let a = readToken() in let b = readToken() in match a,b with |
    LIde(n), LSym(".") when isUpper(n) -> n | LIde(n), LSym(".") ->
    (putError(errUPPERCASE_PRINCIPAL,""); raise ParseError) | LIde(n),
    _ -> (putError(errMISSING_DOT,""); raise ParseError) | _ ->
    (putError(errUNEXPECTED_TOKEN,""); raise ParseError)

let getIndex =
  function () ->
    let a = readToken() in match a with | LIde(n) ->
        let b = readToken() in if b<>LSym(":") then
        (putError(errMISSING_COLON,""); raise ParseError) else n
    | _ -> (putError(errUNEXPECTED_TOKEN,""); raise ParseError);;


let getIndexedDatum =
  let getNum =
    function st -> try (int_of_string(st)) with _ -> -1
  in let splitName =
    function ws ->
      let sl = String.length ws in let pl = (try ((String.rindex ws
```

```
      '_' )) with Not_found -> -1) in if (pl = -1 || pl=sl-1) then
      (putError(errMISSING_INDEX,""); raise ParseError) else (pl, String.sub
      ws 0 pl, String.sub ws (pl+1) (sl-pl-1))
  in function s ->
    let nl = String.length s in let (pos, n, i) = splitName(s) in let im =
    String.sub s (pos+1) (nl-pos-2) in if (isUpper(n)) then (
      match s.[nl-1] with | '+' ->(
          let vi = getNum(im) in if vi<> -1 then PNp(n,vi) else PNp(n^im,0)
         )
      | '-' ->(
          let vi = getNum(im) in if vi<> -1 then PNm(n,vi) else PNm(n^im,0)
         )
      | _ ->(
          let vi = getNum(i) in if vi<> -1 then PN(n,vi) else PN(n^i,0)
         )
     )
    else (
      if (s.[0]<>'n' && s.[0]<>'k') then (
        match s.[nl-1] with | '+' ->(
            let vi = getNum(im) in if vi<> -1 then (
              let (x, nn, np) = splitName(n) in Publ(Var(nn,(np,vi)))
             )
            else Publ(Var(n,(im,0)))
           )
        | '-' ->(
            let vi = getNum(im) in if vi<> -1 then (
              let (x, nn, np) = splitName(n) in Priv(Var(nn,(np,vi)))
             )
            else Priv(Var(n,(im,0)))
           )
        | _ ->(
            let vi = getNum(i) in if vi<> -1 then (
              let (x, nn, np) = splitName(n) in Var(nn,(np,vi))
             )
            else Var(n,(i,0))
           )
       )
      else (
        let vi = getNum(i) in if vi<> -1 then (
          if (s.[0]='k') then K(n,vi) else NO(n,vi)
         )
        else (
          if (s.[0]='k') then K("_"^i^"_"^n,0) else NO("_"^i^"_"^n,0)
         )
       )
     )


let getVar =
  function s ->
    match getIndexedDatum(s) with | Var(p) -> Var(p) | PN(p,i) -> PN(p,i)
    | _ -> (putError(errWRONG_VAR,""); raise ParseError)
```

```
let rec getMessage =
  function() ->
    let a = readToken() in match a with | LSym("(") -> getCouple() |
    LSym("{") -> getCrypt() | LIde(v) -> (pushback(a); getVal()) | _ ->
    (putError(errUNEXPECTED_TOKEN,""); raise ParseError)
and getCouple =
  function () ->
    let m1 = getMessage() in let v1 = readToken() in if v1<>LSym(",") then
    (putError(errMISSING_COMMA,""); raise ParseError) else
      let m2 = getMessage() in let v2 = readToken() in if v2<>LSym(")")
      then (putError(errMISSING_CLOSING_ROUND_PAR,""); raise ParseError)
      else Couple(m1,m2)
and getCrypt =
  function () ->
    let m1 = getMessage() in let v1 = readToken() in if v1<>LSym("}") then
    (putError(errMISSING_CLOSING_CURLY_PAR,""); raise ParseError) else
      let k1 = getVal() in Crypt(m1,k1)

and getVal =
  function () ->
    let a = readToken() in match a with | LIde(s) -> (
        let k = readToken() in match k with | LSym("+") ->
        getIndexedDatum(s^"+") | LSym("-") -> getIndexedDatum(s^"-") | _ ->
        (pushback(k); getIndexedDatum(s))
       )
    | _ -> (putError(errEXPECTED_IDE,""); raise ParseError);;



let pvalue =
  function p ->
    match p with | PNot -> 0 | PAnd | POr -> 1 | Impl -> 2 | DImpl -> 2 |
    Fa(i,n) -> 3 | Ex(i,n) -> 3 | _ -> -1



let reducer =
  function (stack, pr) ->
    match stack with | Pexp(e)::PNot::st -> Pexp(Not(e))::st |
    Pexp(e1)::PAnd::Pexp(e2)::st when pvalue(pr)>=pvalue(PAnd)
    -> Pexp(And(e2,e1))::st | Pexp(e1)::POr::Pexp(e2)::st
    when pvalue(pr)>=pvalue(POr) -> Pexp(Or(e2,e1))::st |
    Pexp(e1)::Impl::Pexp(e2)::st when pvalue(pr)>=pvalue(Impl) ->
    Pexp(Or(Not(e2),e1))::st | Pexp(e1)::DImpl::Pexp(e2)::st when
    pvalue(pr)>=pvalue(DImpl) -> Pexp(Or(And(Not(e2),Not(e1)),And(e1,e2)))::st
    | Pexp(e)::Fa(i,n)::st when pvalue(pr)>=pvalue(Fa(i,n))
    -> Pexp(Forall(i,n,e))::st | Pexp(e)::Ex(i,n)::st when
    pvalue(pr)>=pvalue(Ex(i,n)) -> Pexp(Exists(i,n,e))::st | _ -> raise
    CantReduce;;

let rec parseExp =
  function (tok, stack) ->
    match (tok, stack) with | LIde("true"), _ -> Pexp(True)::stack |
    LIde("false"), _ -> Pexp(False)::stack | LIde("forall"), _ ->
```

```
        let n = getName() in let i = getIndex() in Fa(i,n)::stack
    | LIde("exists"), _ ->
        let n = getName() in let i = getIndex() in Ex(i,n)::stack
    | LSym("&"), _ | LIde("and"), _ -> (try parseExp (tok, reducer(stack,
    PAnd)) with CantReduce -> PAnd::stack) | LSym("|"), _ | LIde("or"), _ ->
    (try parseExp (tok, reducer(stack, POr)) with CantReduce -> POr::stack) |
    LSym("=>"), _ | LIde("implies"), _ -> (try parseExp (tok, reducer(stack,
    Impl)) with CantReduce -> Impl::stack) | LSym("<=>"), _ | LIde("iff"),
    _ -> (try parseExp (tok, reducer(stack, DImpl)) with CantReduce ->
    DImpl::stack) | LSym("!"), _ | LIde("not"), _ -> PNot::stack | LSym(":>"),
    _ | LIde("derive"), _ -> let ms = getMessage() in Pexp(Derive(ms))::stack
    | LSym("="), Token(s)::st | LIde("equal"), Token(s)::st -> let ms =
    getMessage() in Pexp(Equal(getVar(s),ms))::st | LSym("<>"), Token(s)::st
    -> let ms = getMessage() in Pexp(Diff(getVar(s),ms))::st

    | LIde(s), _ ->  Token(s)::stack

    | LSym("("), _ -> Plp::stack | LSym(")"), Pexp(e)::Plp::st -> Pexp(e)::st
    | LSym(")"), _ -> parseExp (tok, reducer(stack, Fa("",""))) | LSym(";"),
    _ -> End::stack | LStop, _ -> (putError(errMISSING_SEMICOLON,""); raise
    ParseError) | _ -> putError(errUNKNOWN_TOKEN,""); raise ParseError;;


let rec reduce_all =
        function stack ->
                match stack with
                        | [Pexp(e)] -> e | [] ->
                        (putError(errMISSING_SEMICOLON,""); raise ParseError)
                        | _ -> reduce_all ( reducer (stack, Fa("","")));;


let rec parser =
    function stack ->
      let a = readToken() in let d =
        (match a with | LSym("=") ->
            (let b = readToken() in if b<>LSym(">") then (pushback(b); a)
            else LSym("=>"))
        | LSym(":") ->
            (let b = readToken() in if b<>LSym(">") then (pushback(b); a)
            else LSym(":>"))
        | LSym("<") ->
            (let b = readToken() in if (b=LSym(">")) then LSym("<>") else if
            (b<>LSym("=")) then (pushback(b); a) else (
              let c = readToken() in if (c=LSym(">")) then LSym("<=>") else
              (pushback(c); LSym("<="))
             )
            )
        | _ -> a) in
      let ss = parseExp (d, stack) in let c = List.hd ss in if c=End then
      stack else parser ss;;


let parseFormula =
  function () ->
```

```
    let (f,ff) =
      try  (true, reduce_all (parser [])) with | CantReduce ->
      (putError(errMALFORMED_EXPR,""); (false,False)) | ParseError ->
      (false, False)

    in if f then ff else False


let rec parseKb =
  function () ->
    let m = getMessage() in let a = readToken() in match a with
    | LSym(",") -> m::(parseKb()) | LSym(";") -> [m] | _ ->
    (putError(errMISSING_SEMICOLON,""); raise ParseError)

let init_parser =
        function fname -> init_tokenizer(fname);;
let done_parser =
        function () -> done_tokenizer();;
```

Principal

```
open Process open Message open Utils


type principal = string*int*(variable list)*process

let create =
  function (name, index, vlist, expr) ->
    let rec updateVars =
      function ls ->
        match ls with | x::xs -> setVariable((name,index),x)::(updateVars xs)
        | [] -> []
    in let rec updateMsg =
      function m ->
        match m with | Couple(m1,m2) -> Couple(updateMsg m1, updateMsg
        m2) | Crypt(m1,m2) -> Crypt(updateMsg m1, updateMsg m2) |
        (Var(v)) -> (Var(setVariable((name,index),v))) | (BVar(v))
        -> (BVar(setVariable((name,index),v))) | Publ(Var(v)) ->
        Publ(Var(setVariable((name,index),v))) | Priv(Var(v)) ->
        Priv(Var(setVariable((name,index),v))) | (NO(s,p)) -> (NO(s,index))
        | (PNp(s,p)) -> (PNp(s,index)) | (PNm(s,p)) -> (PNm(s,index)) |
        (PN(s,p)) -> (PN(s,index)) | (K(s,i)) -> (K(s,index)) | o -> o
    in let rec updateExpr =
      function e ->
        match e with | Par(e1,e2) -> Par(updateExpr e1, updateExpr e2)
        | Sum(e1,e2) -> Sum(updateExpr e1, updateExpr e2) | In(m,e1) ->
        In(updateMsg m, updateExpr e1) | Out(m,e1) -> Out(updateMsg m,
        updateExpr e1) | Nil -> Nil
    in (name, index, updateVars vlist, updateExpr expr)
```

```
let check =
  function (pn,pi,pv,pp) ->
    Process.validate pp

let substitute =
  function asg ->
    function (pn,pi,pv,pp) ->
      let rec subvar =
        function vv ->
          match vv with | v::vs -> if (asg (Var(v)))<>Var(v) then subvar vs
          else v::(subvar vs) | [] -> []
      in (pn,pi, subvar pv, Process.substitute asg pp)

let typeOf =
  function (pn,pi,pv,pp) ->
    function v ->
      let rec findtype =
        function pr ->
          match pr with | In(m,e) ->
              let tt = (typeOfVar m v) in if tt  = Gen then findtype e else tt
          | Out(m,e) ->
              let tt = (typeOfVar m v) in if tt  = Gen then findtype e else tt
          | Par(e1,e2) ->
              let tt1 = findtype e1 in let tt2 = findtype e2 in
              mintype(tt1,tt2)
          | Sum(e1,e2) ->
              let tt1 = findtype e1 in let tt2 = findtype e2 in
              mintype(tt1,tt2)
          | Nil -> Gen
      in if exists pv (function vv -> v=vv) then findtype pp else Gen


let isConsumed =
  function (_,_,v,e) -> (e=Nil && v=[])

let getName =
  function (n,i,_,_) -> (n,i)

let getVars =
  function (_,_,vs,_) -> map vs (function v -> Var(v))

let len =
  function (_,_,_,e) -> Process.len e

let prnt =
  function p ->
    function asg ->
      let (name, index, vlist, expr) = substitute asg p in (
      print_string(name^"_"^string_of_int(index)^": "); print_string("(");
      Message.prntVarlist vlist; print_string(") ["); Process.prnt (expr);
      print_string("]")

     )
```

```
let prntHtml =
  function p ->
    function asg ->
      let (name, index, vlist, expr) = substitute asg p in (
      print_string(name^"<sub>"^string_of_int(index)^"</sub>: ");
      print_string("("); Message.prntVarlist vlist; print_string(") [");
      Process.prntHtml (expr); print_string("]")

     )

let tostring =
  function p ->
    function asg ->
      let (name, index, vlist, expr) = substitute asg p in
      (""^name^"_"^string_of_int(index)^": (")^((Message.tostringVarList
      vlist)^") ["^(Process.tostring (expr))^"]")
```

Princpool

```
open Utils open Principal

exception NoMorePrinc

type ppool = (principal list) * int


let create = function p  -> function n -> (p,n)


let getmax =
        function (pp,max) -> max

let howmany =
        function (pp,max) -> Utils.len pp

let getprinc =
        function (pp,max) -> function n -> function i ->
                if i>max then raise NoMorePrinc else
                        try (
                                let (name,index,vlist,e) = select pp n in
                                        Principal.create (name, i, vlist, e)
                        ) with UtilException -> raise NoMorePrinc
```

Process

```
open Message open Assignment
```

```
exception SubstitutionException

type process =
        | In of msg*process | Out of msg*process | Sum of process*process |
        Par of process*process | Nil

let rec substitute =
        function asg ->
                function e ->
                        match e with
                                | Par(e1, e2) -> Par(substitute (asg)
                                e1, substitute (asg) e2) | Sum(e1, e2) ->
                                Sum(substitute (asg) e1, substitute (asg) e2)
                                | In(m,e1) -> In(asg m, substitute (asg) e1)
                                | Out(m,e1) -> Out(asg m, substitute (asg)
                                e1) | Nil -> Nil


let rec validate =
        function e ->
                match e with
                        | Par(e1, e2) | Sum(e1, e2) -> (validate e1)&&(validate
                        e2) | In(m,e1) | Out(m,e1) -> (validate e1)&&(checkmsg
                        m) | Nil -> true

let rec len =
        function e ->
                match e with
                        | Par(e1, e2) -> (len e1) + (len e2) | Sum(e1, e2) ->
                                let l1 = len e1 in let l2 = len e2 in
                                        if (l1>l2) then l1 else l2
                        | In(m,e1) | Out(m,e1) -> 1+(len e1) | Nil -> 0

let rec tostring =
  function s ->
    match s with | In(m,Nil) -> ("In(")^(Message.tostring m)^")" | Out(m,Nil)
    -> ("Out(")^(Message.tostring m)^")" | In(m,e) -> ("In(")^(Message.tostring
    m)^")."^(tostring e) | Out(m,e) -> ("Out(")^(Message.tostring
    m)^")."^(tostring e) | Sum(e1,e2) -> "("^(tostring e1)^" + "^(tostring
    e2)^")" | Par(e1,e2) -> ("(")^(tostring e1)^" | "^(tostring e2)^")" |
    Nil -> ""

let rec prnt =
  function s ->
    match s with | In(m,Nil) -> print_string(("In(")^(Message.tostring m)^")")
    | Out(m,Nil) ->  print_string(("Out(")^(Message.tostring m)^")") | In(m,e)
    ->  print_string(("In(")^(Message.tostring m)^")."^(tostring e)) | Out(m,e)
    ->  print_string(("Out(")^(Message.tostring m)^")."^(tostring e)) |
    Sum(e1,e2) ->  print_string("("^(tostring e1)^" + "^(tostring e2)^")") |
    Par(e1,e2) ->  print_string(("(")^(tostring e1)^" | "^(tostring e2)^")")
    | Nil -> ()

let rec prntHtml =
  function s ->
```

```
match s with | In(m,Nil) -> print_string(("<i>in</i>("));
prntMsgHtml m; print_string (")") | Out(m,Nil) ->
print_string(("<i>out</i>(")); prntMsgHtml m; print_string (")") | In(m,e)
->  print_string(("<i>in</i>(")); prntMsgHtml m; print_string (").");
prntHtml e | Out(m,e) ->  print_string(("<i>out</i>(")); prntMsgHtml m;
print_string ("). "); prntHtml e | Sum(e1,e2) ->  print_string("(");
prntHtml e1; print_string(" + "); prntHtml e2; print_string(")") |
Par(e1,e2) ->  print_string("("); prntHtml e1; print_string(" | ");
prntHtml e2; print_string(")") | Nil -> ()
```

Status

```
open Context open Assignment open Knowledge open Message

type status = context*assignment*kb

let create =
        function (ct, asg, k) -> (ct, asg, k)

let isTerminal =
        function (ct,asg,k) -> (Context.isEmpty ct)&&(not (Assignment.isVoid
        asg))

let isPartiallyTerminated =
        function (ct,asg,k) -> (Context.hasEmptyPrincipal ct)&&(not
        (Assignment.isVoid asg))

let prnt =
        function (ct,asg,kb) ->
                print_string("Status:\n{\nContext:\n"); Context.prnt ct
                asg; print_string("\nAssignment:\n"); Assignment.prnt
                asg; print_string("\nKnowledge:\n"); Knowledge.prnt kb;
                print_string("\n}\n")
```

Step

```
open Node open Context open Status open Process open Principal open Message
open Assignment open Utils open Action open Core

let findPrincipalRedexes =
  function kman ->
    function k ->
      function asg ->
```

```
        function e ->
          function act ->
            function chs ->
              let rec expand =
                function rds ->
                  function ex -> (
                    match rds with | [] -> [] | (a,(nm, ii, vlist, expr))::xs
                    when expr=Nil -> (a,(nm, ii, vlist, ex))::(expand xs ex)
                    | (a,(nm, ii, vlist, expr))::xs  -> (a,(nm, ii, vlist,
                    Par(expr,ex)))::(expand xs ex)
                   )
              in let rec scan =
                function exp ->
                  match exp with | Par(e1,e2) ->
                      let re1 = scan e1 in let re2 = scan e2 in (expand re1
                      e2) @ (expand re2 e1)
                  | Sum(e1,e2) ->
                      let re1 = scan e1 in let re2 = scan e2  in re1@re2
                  | Nil -> [] | ee when chs(ee) -> (act ee e kman k asg) |
                  _ -> []
              in let (nm,ii,vlist,expr) = e in scan expr


let lazyaction =
  function e ->
    function p ->
      function kman ->
        function k ->
          function asg ->
            let ll = l kman asg in let mm = mu kman k in let (nm,ii,vlist,expr)
            = p in match e with | In(m,e1) ->
                (
                 let jjj = [] (*llazy k asg m m*) in let jjf = filter jjj
                 (function s -> try (Principal.check (Principal.substitute
                 s p)) with _ -> false) in let zzz = if jjf=[] then
                 [Assignment.void()] else jjf in let rrr = debox (map zzz
                 (function s -> let ls = ll (s m) in mul [s] ls (function
                 a -> function b -> a * b))) in let nss = if rrr = []
                 then [Assignment.void()] else rrr in let res = (map nss
                 (function sigma -> mm (sigma) (sigma m))) in let mlm
                 = rebuild nss res (function sigma -> function (m1,a)
                 ->(Input(((nm,ii),m1,sigma*a)),(nm, ii, vlist, e1))) in
                 remclone mlm (function (x,p1) -> function (mes,p2) -> not
                 (Action.equal x mes))
                )

            | Out(m,e1) ->
                (
                 let jjj = [] in let jjf = filter jjj (function s -> try
                 (Principal.check (Principal.substitute s p)) with _ ->
                 false) in let rrr = if jjf=[] then [Assignment.void()] else
                 jjf in let res = debox (map rrr (function s -> let ls = ll
                 (s m) in mul [s] ls (function a -> function b -> a * b)))
                 in let trr = map res (function sigma -> (Output(((nm,ii),m,
```

```
                 sigma)),(nm, ii, vlist, e1))) in remclone trr (function
                 (x,p1) -> function (mes,p2) -> not (Action.equal x mes))
                )

let isOUT =
  function x ->
    match x with | Out(_) -> true | _ -> false
let isIN =
  function x ->
    match x with | In(_) -> true | _ -> false


let getRedex =
  function node ->
    let np = Node.getnd node in Node.getRedex node


let generatelazyRedexesOut =
  function kman ->
    function k ->
      function asg ->
        function ct ->
          function np ->
            let op = (Context.getprincipal (ct, np)) in let pp =
            Principal.substitute asg op in map (((findPrincipalRedexes kman k
            (asg) pp lazyaction isOUT))) (function (b,c) -> (op,b,c))

let generatelazyRedexesIn =
  function kman ->
    function k ->
      function asg ->
        function ct ->
          function np ->
            let op = (Context.getprincipal (ct, np)) in let pp =
            Principal.substitute asg op in map (((findPrincipalRedexes kman k
            (asg) pp lazyaction isIN))) (function (b,c) -> (op,b,c))

let rec transition =
  function frout ->
    function frin ->
      function kman ->
        function node ->
          let (oldct, oldasg, oldkb) = Node.getstatus node in let nd =
          Node.getnd node in if nd = -1 then Node.nulNode() else
            (
             let red = getRedex node in match red  with | (_, Closed, _) ->
             (Node.setnd (node,-1); Node.nulNode()) | (_, NoMoreAct, _) -> (
                 if (nd<(Context.howmany oldct)&&(Node.isOut node)) then (
                   Node.setnd (node,nd+1); let outs = frout kman oldkb oldasg
                   oldct (nd+1) in if outs = [] then (
                     if (nd+1 = (Context.howmany oldct)) then (
                       Node.setnd(node,0); Node.setIn node
                      ) else ();
                     transition frout frin  kman node
```

```
                  )
                 else (
                   Node.close node; Node.settable (node, (outs)); transition
                   frout frin kman node
                  )
               )
              else
                if (nd<(Context.howmany oldct)&&( not (Node.isOut node)))
                then (
                   Node.setnd (node,nd+1); let ins = frin kman oldkb oldasg
                   oldct (nd+1) in (
                    Node.settable (node, (ins)); transition  frout frin
                    kman node
                   )
                  )
                else (
                   Node.setnd (node,-1); Node.nulNode()
                  )
             )
           | (p1, Output(ppn,m,sigma), p2) -> (
               let newasg = oldasg * sigma in let newct = Context.applyall
               (oldct ++ (p1, (Principal.substitute newasg p2))) newasg in
               let newkb = (Kmanager.expandK(kman, oldkb, newasg, sigma m))
               in let newst = Status.create(newct, newasg, newkb) in let
               newnode = Node.create(newst) in ( Node.setfather (newnode,
               node); Node.setlabel (newnode, Output(ppn,m,sigma)); newnode)
             )
           | (p1, Input(ppn,m,sigma), p2) -> (
               let newasg = oldasg * sigma in let newct = Context.applyall
               (oldct ++ (p1, (Principal.substitute newasg p2))) newasg
               in let newkb = Kmanager.updateK(kman,oldkb,newasg) in let
               newst = Status.create(newct, newasg, newkb) in let newnode
               = Node.create(newst) in ( Node.setfather (newnode, node);
               Node.setlabel (newnode, Input(ppn,m,sigma)); newnode)
             )
         )

let getTransitionFunction =
  function km ->
     transition generatelazyRedexesOut generatelazyRedexesIn km
```

Thestack

```
open Utils

exception StackException type 'a stack = 'a list


let init =
        function bottom -> [bottom]
```

```
let push =
        function ns -> function n ->
                n::ns

let pushl =
        function ns -> function nl -> nl@ns

let pushlr =
        function ns -> function nl -> reverse(nl)@ns

let pop =
        function ns ->
                if ns=[] then raise StackException else tail ns
let top =
        function ns ->
                if ns=[] then raise StackException else head ns

let isEmpty =
        function ns ->
                ns = []

let length =
  function ns ->  len ns;
```

Tokenizer

```
type ltok =
        | LIde of string | LSym of string | LEnd | LStop | LUnknown of char;;

type lbuf = string*(int ref)*int;;

let forward buf = match buf with
        | (s,n,sz) -> n:= !n+1;;

let nextIs = fun (ch, (s,n,sz)) ->
        if !n=sz-1 then false else ch=s.[!n+1];;


let getIde buf = let (s,n,sz) = buf in
        let rec cntIde l = if l=sz then l
                else(
                        match s.[l] with
                                | 'A'..'Z' | 'a'..'z' | '0'..'9' | '_' ->
                                cntIde (l+1) | _  -> l
                        )
        in let k = cntIde !n
                in let res = String.sub s !n (k - !n)
                        in (n:=k; res);;

let rec gettok buf =
        let analyze ch = (
```

```
                match ch with
                        | ' ' | '\t' | '\n' -> (forward buf; gettok buf) |
                        'A'..'Z' | 'a'..'z' | '0'..'9' -> LIde(getIde buf)
                        | '(' | ')' | '.' | '|' | '+' | '{' | '}' | ',' |
                        '?' | ';' | '-' | ':' | '[' | ']' | '&' | '|' |
                        '>' | '=' | '<' | '!'  | '/' | '*' -> (forward
                        buf; LSym(String.make 1 ch)) | _ -> (forward buf;
                        LUnknown(ch))
                )
        in match buf with
                | (s,n,sz)  when !n<sz-> analyze s.[!n] | (s,n,sz) -> LEnd;;

let prnt h = match h with
        | LIde(s) -> print_string ("-->"^s^"\n") | LSym(s) ->
        print_string ("-->"^s^"\n") | LEnd -> print_string ("fine\n") |
        LStop -> print_string ("stop\n") | LUnknown(c) -> print_string
        ("Unknown:"^(String.make 1 c)^"\n");;


let mainbuf = ref(("",ref(0),0));; let chan = ref(stdin);; let cline = ref(0);;
let init_tokenizer fname = chan:= (open_in fname);; let done_tokenizer =
fun () -> close_in !chan;;


let flushline = fun () ->
        let nline = try (input_line !chan) with End_of_file -> "\n" in
                if (nline="\n") then (cline:=!cline+1; mainbuf:= (nline,
                ref(0), -1)) else (cline:=!cline+1; mainbuf:= (nline, ref(0),
                String.length nline));;

let rec getToken = function () ->
        let res = gettok !mainbuf
                in match res with
                        | LEnd -> (
                                flushline(); let (nline,n,sz) = !mainbuf in
                                        if sz<> -1 then getToken() else LStop
                                )
                        | _ -> res;;

let pushback s = match s with
        | LIde(k) | LSym(k) -> let (ss, p, nn) = !mainbuf in
        mainbuf:=(ss,ref(!p-String.length k), nn) | _ -> ();;

let where = fun () -> let (nline,n,sz) = !mainbuf in (!cline,!n);;
```

Tracer

```
open Message open Principal open Kmanager open Node open Parsecmdline open
Logic open Step open Utils open Thestack open Action open Verifier open
```

```
Csolver open Join

exception StopSearch exception AttackFound exception NextTrace

type ss = {
    mutable bfm: float; mutable maxbf: int; mutable minbf: int; mutable
    len_traces: int list; mutable son_traces: float list;

    mutable initial_nodes: int; mutable current_node: int; mutable
    closed_nodes: int; mutable transitions: int; mutable num_nodes: int;
    mutable num_traces: int; mutable num_terminal_traces: int; mutable
    num_attacks: int;

    mutable root_node: node; mutable startnodes: node list; mutable startjoins:
    (node * cs)  list list; mutable startpls: npls list; mutable joinpls:
    npls list;

    mutable kman: km; mutable trans:  node -> node; mutable init:  node ->
    node; mutable join: node -> (node) list; mutable opts: options;

    mutable jointime: float; mutable searchtime: float; mutable verifytime:
    float; mutable elapsed: float;

    mutable vflag: bool;
  }


let init_search =
  function prns ->
    function kbl ->
      function options ->
        let ct = Context.void() in let asg = Assignment.void() in let kmanage
        = Kmanager.initKM (Sys.max_array_length) in let kb1 = (Kmanager.newK
        kmanage) in let pool = Princpool.create (prns) options.max_princ in
        let transition = getTransitionFunction kmanage in let initjoinfn =
        initjoin pool kmanage options.ic in let joinmodel = joincontext
        kmanage options.greed in let kb = (Kmanager.expandK(kmanage, kb1,
        asg,(   fold2 kbl (function a -> function b -> Couple(a,b))  )))
        in let st = Status.create(ct,asg,kb) in let root = Node.create(st)
        in let current_status = {
          bfm = 0.0; maxbf = 0; minbf = 0; len_traces = []; son_traces = [];
          initial_nodes = getinitialnodes (len prns) (options.max_princ);
          current_node = 0; closed_nodes = 0; transitions = 0; num_nodes
          = 0; num_traces = 0; num_terminal_traces = 0; num_attacks = 0;
          root_node = root; startnodes = []; startjoins = []; startpls =
          []; joinpls = []; kman = kmanage; trans = transition; init =
          initjoinfn; join = joinmodel; opts = options; jointime = 0.0;
          searchtime = 0.0; verifytime = 0.0; elapsed = 0.0; vflag = false;
        } in current_status

let format_time =
  function time ->
    let hour = int_of_float(ceil time) / 3600 in let min = (int_of_float(ceil
    time) - (hour * 3600)) / 60 in let sec = (int_of_float(ceil time)
```

```
    - (3600*hour + 60*min)) in if (hour < 0) then ("infinity") else
    (string_of_int(hour)^":"^string_of_int(min)^":"^string_of_int(sec))


let rec prntTrace =
  function ss ->
    function nn ->
      if Node.isNul(nn) then  print_string("\n-------\n Trace
      ("^string_of_int(ss.num_traces)^")|<<---------------------------|\n-------\n\n")
      else
        let np = Node.getfather nn in (
         prntTrace ss (np); Node.prnt nn
        )

let rec build_intruder =
  function nn ->
    function aaaa ->
      let ff = Node.getfather nn in if Node.isNul(ff) then
      print_string("Intruder:\n") else (
        build_intruder ff aaaa; let acc = Node.getlabel nn in match acc with |
        Output((nm,ii),mm,s) -> print_string(nm^"_"^string_of_int(ii)^" ->
        I: ");prntMsg (aaaa mm); print_string("\n") | Input((nm,ii),mm,s)
        -> print_string("I -> "^nm^"_"^string_of_int(ii)^": ");prntMsg
        (aaaa mm); print_string("\n") | _ -> ()
)

let rec build_intruder_html =
  function nn ->
    function aaaa ->
      let ff = Node.getfather nn in if Node.isNul(ff) then () else (
        build_intruder_html ff aaaa; let acc = Node.getlabel nn in match
        acc with | Output((nm,ii),mm,s) ->
            print_string(nm^"<sub>"^string_of_int(ii)^"</sub> -&gt <b>I</b>:
            "); prntMsgHtml (aaaa mm); print_string("<br>")
        | Input((nm,ii),mm,s) ->
            print_string("<b>I</b> -> "^nm^"<sub>"^string_of_int(ii)^"</sub>:
            "); prntMsgHtml (aaaa mm); print_string("<br>")
        | _ -> ()
)

let calcexpect =
  function ss ->
    let i = ref(-1) in let vals = map (ss.len_traces) (function lt ->
    i:=!i+1; if !i < ss.current_node then 0.0 else  (select ss.son_traces
    !i) *. (ss.bfm ** (float_of_int(lt) ))) in fold vals (function en ->
    function ctot -> en +. ctot) (float_of_int(ss.num_nodes))


let calcbf =
  function ss ->
    function ns ->
      if ns>(ss.maxbf) then ss.maxbf <- ns else (); if
      ns<(ss.minbf)||(ss.minbf=0) then ss.minbf <- ns else (); ss.transitions
      <- ss.transitions+ns; ss.closed_nodes <- ss.closed_nodes+1; ss.bfm <-
```

```
      (float_of_int(ss.transitions)) /. (float_of_int(ss.closed_nodes))

let feedback =
  function ss ->
    if (ss.opts.feedback) then(
      let expect = calcexpect ss in let perc = (100.0
      *. (float_of_int(ss.num_nodes) /. expect)) in let tm = Sys.time()
      -. ss.searchtime in let time = ((tm *. 100.0) /. perc) -. tm
      in let remaining = format_time time in let elapsed = format_time tm in
      prerr_string("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\
      prerr_string(string_of_int(int_of_float(perc))^"%");
      prerr_string(" elapsed "^elapsed^" "); prerr_string("~
      remaining "^remaining^" "); prerr_string(" subtree
      "^string_of_int(ss.current_node+1)^"/"^string_of_int(ss.initial_nodes));
      prerr_string(" ("^string_of_int(ss.num_attacks)^") "); if (ss.vflag)
      then (
        ss.vflag <- false; prerr_string(" - V ")
       ) else (
        prerr_string(" - S ")
       );
      flush stderr
     ) else ()


let printAttack =
  function ss ->
    function nn ->
      function conj ->
        function nid ->
          function cssf ->
            if (ss.opts.attack_type = 2) then(
              ss.num_attacks <- ss.num_attacks+1; print_string("<center><table
              border=1><tr><td colspan=2 >"); print_string("
              <b>ATTACK "^string_of_int(ss.num_attacks)^"
              -- (Id: "^string_of_int(nid)^")</b>");
              print_string("</td></tr><tr><td><b>Violated
              Constraints</b></td><td><b>Open
              Variables</b></td></tr><tr><td>"); let (cti,asgi,ki) =
              Node.getstatus (select ss.startnodes ss.current_node) in let
              (ctx,asgx,kx) = Node.getstatus nn in ( Logic.prntnplHtml
              ([conj]); print_string("</td><td>"); let asgcs =
              Csolver.toasg cssf in let mdls = Csolver.linearize cssf
              in let cpl1 = couplify asgcs mdls in let (ax,cx) = head
              cpl1 in let newasg = (Assignment.comp ax asgx) in let
              (nv,vars) = Context.getOpenVariables cti in print_list
              vars (function v -> Message.prntMsgHtml v; print_string("
              -&gt "); Message.prntMsgHtml (newasg v);print_string("<br>"));
              print_string("</td></tr><tr><td><b>Knowledge</b></td><td><b>Context</b></td></tr>
              Knowledge.prntHtml (Knowledge.applyall kx newasg);
              print_string("</td><td>"); Context.prntHtml
              cti newasg; print_string("</td></tr>");
              print_string("<tr><td><b>Model</b></td><td><b>Intruder
              process<b></td></tr><tr><td>"); Csolver.prntHtml cx;
              print_string("</td><td>"); build_intruder_html nn newasg;
```

```
              print_string("</td></tr></table></center><br><hr><br>")
               )
             ) else (
              ss.num_attacks <- ss.num_attacks+1; print_string("
              ATTACK "^string_of_int(ss.num_attacks)^"
              -- (Id: "^string_of_int(nid)^")");
              print_string("\n--------------------------------------------------------------\n\n");
              let (cti,asgi,ki) = Node.getstatus (select ss.startnodes
              ss.current_node) in let (ctx,asgx,kx) = Node.getstatus nn in
              ( print_string("Violated Constraints:\n"); Logic.prntnpl
              ([conj]); let asgcs = Csolver.toasg cssf in let mdls =
              Csolver.linearize cssf in let cpl1 = couplify asgcs mdls in
              let (ax,cx) = head cpl1 in let newasg = (Assignment.comp
              ax asgx) in print_string("\n***************\n");
              print_string("Open Variables:\n"); let (nv,vars) =
              Context.getOpenVariables cti in print_list vars (function v ->
              Message.prntMsg v; print_string(" -> "); Message.prntMsg (asgx
              v);print_string("\n")); print_string("Context:\n"); Context.prnt
              cti newasg; print_string("Knowledge:\n"); Knowledge.prnt
              (Knowledge.applyall kx newasg); print_string("\nModel:\n");
              Csolver.prnt cx; build_intruder nn newasg;
              print_string("\n\n\n");
              print_string("There are
              "^string_of_int(len cpl1)^" possible models.");
              print_string("\n\n");
              if (ss.opts.mga) then () else (
                print_list (tail cpl1) ( function (aa,cc) ->
                  print_string("\nModel:\n"); Csolver.prnt cc;
               ) ) );
              print_string("\n***************\n");
              print_string("\n--------------------------------------------------------------\n\n");

             );
            flush stdout



let verification =
  function ss ->
    function nn ->
      function cf ->
        let ask =
          function () ->
            if (ss.opts.interactive) then (
              prerr_string("\nAttack found
              ("^string_of_int(ss.num_attacks)^"). [N]ext, [S]top, [O]ther
              trace? "); flush stderr; let res = try (input_line stdin)
              with End_of_file -> "Y" in if (res="O") then (raise NextTrace)
              else if (res="S") then (raise StopSearch) else ()
             ) else ()
        in ss.vflag <- true; feedback ss;

        let conjuncts = select ss.startpls ss.current_node in let jpl =
        select ss.joinpls ss.current_node in try (
```

```
          iter conjuncts ( function ff ->
            ss.vflag <- true; feedback ss; let (conj,verf) = fast_checkpl
            ([ff]) (cf) (ss.kman) nn in try (
              iter verf ( function (pl,asg,ccf) ->
                let (ctn,asgn, kbn) = Node.getstatus nn in let newasg =
                Assignment.comp asgn asg in let newct = Context.applyall
                ctn newasg in let newnode = Node.create((newct,newasg,kbn))
                in Node.setfather (newnode, nn); Node.setlabel (newnode,
                Join(("Lazy Check",Node.getid newnode),newasg)); ss.vflag <-
                true; feedback ss; if (fast_sat_model (jpl) (ccf) (ss.kman)
                (newnode)) then (
                  printAttack ss newnode ff (Node.getid nn) ccf; raise
                  AttackFound
                 )
                else ()
             ) ) with AttackFound -> ask(); if (ss.opts.mga && (not
             ss.opts.interactive)) then raise NextTrace else ();
         ) ) with NextTrace -> ()


let iterstack =
  function ss ->
    function cf ->
      function nstack ->
        let cti = Sys.time() in let top = Thestack.top nstack in let nk =
        ss.trans top in ss.num_nodes <- (if(Node.getid nk > ss.num_nodes) then
        Node.getid nk else ss.num_nodes); if (cti -. ss.elapsed > 1.0 ) then (
          ss.elapsed <- cti; feedback ss
         )
        else (); if (Node.isNul nk) then (
          let nss = Thestack.pop nstack in if (Node.isLeaf top) then (
            ss.num_traces <- ss.num_traces+1; if (Node.isTerminal top ) then (
              ss.num_terminal_traces <- ss.num_terminal_traces +1; if
              (ss.opts.output_type = 2 ) then (
                let stime = Sys.time() in (verification ss top cf);
                ss.verifytime <- ss.verifytime +. Sys.time() -. stime;
               )
              else
                if (ss.opts.output_type = 0) then prntTrace ss top else
                  if (ss.opts.output_type = 1 && (not ss.opts.lazyness))
                  then Gviz.prntTrace top (ss.opts.go) else
                    if (ss.opts.output_type = 1) then (
                       Gviz.prntTrace top (ss.opts.go)
                     )
                    else ();
              nss
             )
            else (
              if (ss.opts.output_type=0 && (not ss.opts.trace_terminal)) then
              prntTrace ss top else (); if (ss.opts.output_type=1 && (not
              ss.opts.trace_terminal)) then Gviz.prntTrace top (ss.opts.go)
              else (); nss
             )
           )
```

```
          else (
            calcbf ss (Node.getSons top); nss
           )
         )
        else (
          Thestack.push nstack nk
         )

let do_joins =
  function ss ->
    function pl ->
      function plj ->
        let nc = ref(0) in let nd = ref(0) in let rec join_principals =
          function () ->
            let nk = (ss.init ss.root_node) in if (Node.isNul nk) then []
            else (
              if (ss.opts.feedback) then (nc:=!nc+1;
              prerr_string("\b\b\b\b\b\b\b\b\b\b"^string_of_int(!nc)^"/"^string_of_int(ss.initial_noc
              "); flush stderr) else (); let ne = (nk, normalize (Not(pl))
              nk, normalize ((plj)) nk ) in ne::(join_principals())
             )
        in ( let rec first_generation_lazy =
          function ls ->
            function js ->
              match ls, js with | nn::nns, pj::pjs ->
                  if (ss.opts.feedback) then (nd:=!nd+1;
                  prerr_string("\b\b\b\b\b\b\b\b\b\b"^string_of_int(!nd)^"/"^string_of_int(ss.initial
                  "); flush stderr) else (); let nodes = ss.join nn in let
                  thenodes = filter nodes (fast_sat pj (ss.kman)) in let
                  res = map thenodes (function n -> (n,Csolver.init()))
                  in res::(first_generation_lazy nns pjs)
              | [], [] ->  []
        in if (ss.opts.feedback) then (
          prerr_string("\n\nJoining principals...\n\n"); flush stderr
         ) else ();

        let stime = Sys.time() in let rb = join_principals() in if
        (ss.opts.feedback) then (
          prerr_string("\n\nFirst generation...\n\n"); flush stderr
         ) else ();
        ss.startnodes <- (map rb (function (nn, pp, pj) -> nn)); ss.startpls <-
        (map rb (function (nn, pp, pj) -> pp)); ss.joinpls <- (map rb (function
        (nn, pp, pj) -> pj)); ss.current_node <- 0; ss.initial_nodes <-
        len ss.startnodes; ss.len_traces <- map ss.startnodes (function nn
        -> let (ctx,asgx,kx) = Node.getstatus nn in Context.getlentrace
        ctx); ss.num_nodes <- ss.initial_nodes +1; ss.transitions <-
        ss.initial_nodes; let ra =
          first_generation_lazy (ss.startnodes) (ss.joinpls)
        in ss.startjoins <- ra; ss.son_traces <- map (ss.startjoins)
        (function l -> float_of_int(len l)); ss.jointime <- Sys.time() -. stime
       )


let do_search =
```

```
  function ss ->
    let rec searchstack =
      function ls ->
        match ls with | (nn,cf)::nns -> (
            let nstck = ref(Thestack.init nn) in (
             while (not (Thestack.isEmpty (!nstck))) do
               (nstck := (iterstack ss cf (!nstck)))
             done
            )); searchstack nns
        | []  -> ()
    in let rec startsearch =
      function ls ->
        match ls with | jn::jns ->
            searchstack jn; ss.current_node <- ss.current_node+1;  feedback
            ss; startsearch jns
        | [] ->  ss.current_node <- ss.current_node-1;  feedback ss
    in let stime = Sys.time() in if (ss.opts.feedback) then (
      prerr_string("\n\nSearching...\n\n"); flush stderr
     ) else ();
    (try (startsearch ss.startjoins) with StopSearch -> ()); ss.searchtime <-
    Sys.time() -. stime




let prnt_stats =
  function chan ->
    function ss ->
      output_string chan
      ("\n\n-----------------------------------------\nTotal traces:
      "^string_of_int(ss.num_traces)^"\n"); output_string chan ("Total terminal
      traces: "^string_of_int(ss.num_terminal_traces)^"\n"); output_string
      chan ("Total nodes: "^string_of_int(ss.num_nodes)^"\n"); output_string
      chan ("Total starting nodes: "^string_of_int(ss.initial_nodes)^"\n");
      output_string chan ("Total pruned starting nodes: "^string_of_int(len
      (filter ss.startjoins (function l -> (len l) = 0)))^"\n");
      output_string chan ("\nTotal time: "^format_time(ss.searchtime
      +. ss.jointime)^"\n"); output_string chan ("Search time:
      "^format_time(ss.searchtime -. ss.verifytime )^"\n"); output_string
      chan ("Join time: "^format_time(ss.jointime)^"\n"); output_string
      chan ("Verify time: "^format_time(ss.verifytime)^"\n"); let nps =
      float_of_int(ss.num_nodes) /. (ss.searchtime -. ss.verifytime)
      in let nts = float_of_int(ss.num_traces) /. (ss.searchtime
      -. ss.verifytime) in let nvs = float_of_int(ss.num_terminal_traces)
      /. (ss.verifytime) in output_string chan ("\nnode/s:
      "^string_of_int(int_of_float(nps))^"\n"); output_string chan
      ("trace/s: "^string_of_int(int_of_float(nts))^"\n"); output_string
      chan ("verification/s: "^string_of_int(int_of_float(nvs))^"\n");
      output_string chan ("branching factor: "^string_of_float(ss.bfm)^",
      ("^string_of_int(ss.transitions)^"/"^string_of_int(ss.closed_nodes)^")\n");
      output_string chan ("max branching factor:
      "^string_of_int(ss.maxbf)^"\n"); output_string chan ("min branching
      factor: "^string_of_int(ss.minbf)^"\n"); output_string chan ("initial
      branching factors: "); print_list (ss.son_traces) (function
```

```
      i -> output_string chan (string_of_int(int_of_float(i))^" "));
      output_string chan ("\ntrace length: "); print_list (ss.len_traces)
      (function i -> output_string chan (string_of_int(i)^" "));
      output_string chan ("\n"); output_string chan ("\nNumber of attacks:
      "^string_of_int(ss.num_attacks)^"\n--------------------------------------------\n")

let print_stats = prnt_stats stdout let prerr_stats = prnt_stats stderr
```

Utils

```
exception UtilException

let rec len =
  function l ->
    match l with | x::xs -> 1 + (len xs) | [] -> 0

let rec select =
  function l ->
    function n ->
      match l with | x::xs when n=0 -> x | x::xs -> select (xs) (n-1) |
      [] -> print_string("select!\n"); raise UtilException

let rec debox =
  function l ->
    match l with | x::xs -> x@(debox xs) | [] -> []

let rec map =
  function l ->
    function f ->
      match l with | x::xs -> (f x)::(map xs f) | [] -> []

let rec iter =
  function l ->
    function f ->
      match l with | x::xs -> let y = (f x) in  (iter xs f) | [] -> ()


let rec filter =
  function l ->
    function p ->
      match l with | x::xs -> if (p x) then x::(filter xs p) else (filter
      xs p) | [] -> []

let rec mul =
  function la ->
    function lb ->
      function f ->
        match la with | a::aa -> (map lb (f a))@(mul aa lb f) | [] -> []

let rec exists =
  function l ->
```

```
    function p ->
      match l with | x::xs -> if (p x) then true else exists xs p | [] -> false

let rec forall =
  function l ->
    function p ->
      match l with | x::xs -> if (p x) then (forall xs p) else false |
      [] -> true

let rec remclone =
  function l ->
    function p ->
      match l with | x::xs -> x::(remclone (filter xs (p x)) p) | [] -> []

let diff =
  function l1 ->
    function l2 ->
      filter l1 (function x -> forall l2 (function y -> y<>x ))


let rec print_list =
  function ll ->
    function pr ->
      match ll with | x::xs -> pr x; print_list xs pr | [] ->
      print_string("\n")



let rec fold =
  function l ->
    function f ->
      function en ->
        match l with | ll::[] -> f ll en | ll::ls -> f ll (fold ls f en) |
        [] ->  print_string("fold!\n");raise UtilException


let rec fold2 =
  function l ->
    function f ->
      match l with | ll1::ll2::[] -> f ll1 ll2 | ll::[] ->
      print_string("fold2_1!\n"); raise UtilException | [] ->
      print_string("fold2_2!\n");raise UtilException | ll::ls -> f ll (fold2
      ls f )


let head =
  function l ->
    match l with | ll::ls -> ll | [] ->  print_string("head!\n");raise
    UtilException

let tail =
  function l ->
    match l with | ll::ls -> ls | [] -> []
```

```
let rec aleph =
  function l ->
    match l with | ll::ls ->
        (let ss = aleph ls in (mul [ll] (aleph ls) (function a -> function
        b -> a::b))@ss )
    | [] -> [[]]



let rec rebuild =
  function l1 ->
    function l2 ->
      function f ->
        match (l1,l2) with | x::xs,y::ys -> (mul ([x]) y f)@(rebuild xs ys f)
        | [],[] -> []



let rec reverse =
  function l ->
    match l with | x::xs -> (reverse xs)@[x] | [] -> []


let rec grow =
  function a ->
    function n ->
      if n<=0 then [] else a::(grow a (n-1))



let rec couplify =
  function l1 ->
    function l2 ->
      match l1,l2 with | x::xs, y::ys -> (x,y)::(couplify xs ys) | [],[] ->
      [] | _ -> print_string("Couplify!\n"); raise UtilException


let rec tensor =
  function ll ->
    function f ->
      match ll with | x::xs ->
          ( (f x)::xs)::(mul [x] (tensor xs f) ( function a -> function b
          -> a::b))
      | [] -> []

let rec partition =
  function elements ->
    let rec nextpart =
      function part ->
        function e ->
          match part with | y::[] -> ([[e]]@y)::(tensor y (function a ->
          e::a)) | y::ys ->
```

```
let newpart = ([[e]]@y)::(tensor y (function a -> e::a)) in
              newpart@(nextpart ys e)
          | [] -> [[[e]]]
    in match elements with | x::xs ->
        let prevp = partition xs in nextpart prevp x
    | [] -> []


let compose =
  function f1 ->
    function f2 ->
      function x ->
        f2 (f1 x)

let invert =
  function f ->
    (
     function x ->
       function y -> f y x
    )

let half =
  function f ->
    (
     function x -> f x x
    )


let rec sort = function l -> function lt -> if l=[] then l else (
 let top = head l in let half1 = filter l (lt top) in let half2 = filter l
 (compose (lt top) (not)) in if (half1=[] || half2=[]) then l else (sort
 half1 lt)@(sort half2 lt)
)
```

Verifier

```
open Logic open Assignment open Core open Utils open Csolver open Message

type ops = Positive | Negative | Noop


let solve_membership =
  function Derive(m) ->
    function kman ->
      function node ->
        let (ct, asg, k) = Node.getstatus node in map (mu_eq kman k asg m)
        (snd)

let solve_equality =
  function Equal(v,m) ->
    function kman ->
```

```
      function node ->
        let (ct, asg, k) = Node.getstatus node in ni_eq kman asg v m



let expandasg =
  function asg ->
    function kman ->
      let lcs = (listfy asg)  in let kx = Kmanager.getK kman 1 in if lcs =
      [] then [] else
        let toexp = fold lcs (
          function (a,b) ->
            function ls ->
              let t=asg a in (
               match a,t with | Var(_), SVar(_,_,ty) when ((ty<>Gen &&
               ty<> GenR))->
                   let res = mu_eq kman kx asg t in (positivize(map res
                   (function (mm,ss) -> (t,mm)))):ls
               | SVar(v,_,ty), _ when ((ty = Gen || ty = GenR) && notSVar(t))
               ->
                   (positivize([(a,t)])):ls
               | _  -> ls
              )
         ) [] in
              remclone toexp ( <> )


let getemptymodel =
  function () -> (Assignment.void(),Csolver.init())


let fast_solveconjunct =
  function pls ->
    function kman ->
      function node ->
        let rec fast_solve_positive_atoms =
          function spls ->
            match spls with | Equal(v,m)::lls ->
                let asgl = solve_equality (Equal(v,m)) kman node in if asgl =
                [] then (False::lls, [], Positive) else (True::lls, asgl,
                Positive)
            | Derive(m)::lls ->
                let asgl = solve_membership (Derive(m)) kman node in if
                asgl = [] then (False::lls, [], Positive) else (True::lls,
                asgl, Positive)
            | ll::lls ->
                let (a,c,d) = fast_solve_positive_atoms lls in (ll::a, c, d)
            | [] -> ([],[], Noop)
        in let rec fast_solve_negative_atoms =
          function spls ->
            match spls with | Diff(v,m)::lls ->
                let asgl = solve_equality (Equal(v,m)) kman node in if asgl =
                [] then (True::lls, [], Negative) else
                  if (Assignment.isVoid (head asgl)) then (False::lls,
```

```
                [], Negative) else (True::lls, asgl, Negative)
            | NDerive(m)::lls ->
                let asgl = solve_membership (Derive(m)) kman node in if asgl =
                [] then (True::lls, [], Negative) else
                  if (Assignment.isVoid (head asgl)) then (False::lls,
                  [], Negative) else (True::lls, asgl, Negative)
            | ll::lls ->
                let (a,c,d) = fast_solve_negative_atoms lls in (ll::a, c, d)
            | [] -> ([],[], Noop)
        in

        let pres = (fast_solve_positive_atoms pls) in match pres  with |
        (_,_,Noop) -> fast_solve_negative_atoms pls | _ -> pres


let fast_solve =
  function stack ->
    function kman ->
      function node ->
        let (pl,pasg,css) = Thestack.top stack in let cs = preserve_truth
        css in if (check cs) then (([], pasg, cs),Thestack.pop stack) else  (
          let (newpl1, asg, tops) = fast_solveconjunct pl kman node in if
          (tops=Noop) then (
            (*verification terminated*) ((newpl1, pasg, cs),(Thestack.pop
            stack))
           )
          else (
            if (asg=[]) then (
              (*Unification failed -> unicity of result*) (([],pasg,cs),
              Thestack.push (Thestack.pop stack) ((newpl1, pasg, cs)))
             ) else (
              (*New constraints -> concretizing results*) if (tops=Positive)
              then (
                (*concretizing with assignments*) let newpstack = map asg (
                  function x ->
                    let lll = positivize(listfy x) in (debox (concretize
                    ([newpl1]) x), pasg*x, addconstraints lll cs)
                 )
                in (([],pasg,cs), Thestack.pushl (Thestack.pop stack)
                newpstack)
               ) else (
                let newfstack = (newpl1, pasg, (addconjuncts (map asg
                (function x -> negativize (listfy x))) cs))

                in (([],pasg,cs), Thestack.push (Thestack.pop stack) newfstack)
               )
             )
           )
         )


let fast_solver =
  function bottom ->
    function kman ->
```

```
      function node ->
        let pstack = ref (Thestack.init bottom) in let result = ref ([]) in (
         while (not (Thestack.isEmpty (!pstack))) do
           (
            let (tres, tstack) = (fast_solve (!pstack) kman node) in pstack
            := tstack; match tres with | ([],_,_) -> () | _ -> result :=
            (tres)::(!result)
           )
         done; !result
        )


let fast_checkpl =
  function pl ->
    function css ->
      function kman ->
        function node ->
          let (ct,asg,k) = Node.getstatus node in let npl =
          concretize pl asg in let lcs = positivize (listfy asg)
          in let csf = Csolver.adddisjuncts (expandasg asg kman) css
          (*(Csolver.addconstraints lcs css)*)  in let rec scan_conjuncts =
            function ipl ->
              function pll -> match ipl,pll with | ipls::iplls, pls::plls ->
                  let res = fast_solver ((pls,asg,csf)) kman node in let ok =
                  filter res (function (lp,lasg,lcsf) -> (forall lp (function
                  x -> (x=True))) && (not (Csolver.check csf))) in if (ok =
                  []) then scan_conjuncts iplls plls else (ipls,map ok
                  (function (lp,lasg,lcsf) -> ([lp],lasg,lcsf)))
              | [],[] -> ([],[])
          in scan_conjuncts pl npl


let very_fast_checkpl =
  function pl ->
    function css ->
      function kman ->
        function node ->
          let (ct,asg,k) = Node.getstatus node in let npl = concretize pl
          asg in let csf = Csolver.adddisjuncts (expandasg asg kman) css in
          let rec scan_conjuncts =
            function ipl ->
              function pll -> match ipl,pll with | ipls::iplls, pls::plls ->
                  let res = fast_solver ((pls,asg,csf)) kman node in let ok =
                  filter res (function (lp,lasg,lcsf) -> (forall lp (function
                  x -> (x=True))) && (not (Csolver.check csf))) in if (ok =
                  []) then scan_conjuncts iplls plls else (ipls,map ok
                  (function (lp,lasg,lcsf) -> ([lp],lasg,lcsf)))
              | [],[] -> ([],[])
          in scan_conjuncts pl npl


let fast_sat_model =
  function pl ->
    function cs ->
```

```
      function kman ->
        function node ->
          let res = very_fast_checkpl pl (cs) kman node in (res <> ([],[]) )


let fast_sat =
  function pl ->
    function kman ->
      function node ->
        fast_sat_model pl (Csolver.init()) kman node
```