Bond University

MASTER'S THESIS

Automated Analysis of Industrial Scale Security Protocols.

Plasto, Daniel

Award date:
2005

Automated Analysis of Industrial Scale Security Protocols

by

Daniel J. Plasto

A thesis submitted to the

Faculty of Information Technology
Bond University

for the degree of

MASTER OF SCIENCE (COMPUTER SCIENCE)

December 2004

Abstract

Security protocols provide a communication architecture upon which security-sensitive distributed applications are built. Flaws in security protocols can expose applications to exploitation and manipulation. A number of formal analysis techniques have been applied to security protocols, with the ultimate goal of verifying whether or not a protocol fulfils its stated security requirements. These tools are limited in a number of ways. They are not fully automated and require considerable effort and expertise to operate. The specification languages often lack expressiveness. Furthermore the model checkers often cannot handle large industrial scale protocols due to the enormous number of states generated.

Current research is addressing many of the limitations of the older tools by using state-of-the-art search optimisation and modelling techniques. This dissertation examines new ways in which industrial protocols can be analysed and presents abstract communication channels; a method for explicitly specifying assumptions made about the medium over which participants communicate.

Statement of originality

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Daniel J. Plasto

Acknowledgments

First and foremost I would like to thank Dr. Padmanabahn Krishnan and Dr. Jorge Cuellar. Dr. Krishnan directed my research and initiated my internship at Siemens. In particular he critiqued my thesis and gave valuable feedback on a number of drafts. I am grateful for his assistance and my thesis is the better for it. Dr. Cuellar provided direction, ideas and discussion, as well as helping me to understand the concepts behind HLPSL and TLA. My thanks go to Dr. Cuellar for my memorable time at Siemens.

The AVISPA team was particularly helpful to me. In particular I would like to thank Luca Compagna for his help with SATMC and IF, and for our fun discussions on channels and compositionality. Sebastian Moedersheim was extremely helpful to me while learning about the HLPSL and IF languages, and was always quick to provide answers to my questions.

My colleagues at Siemens made my stay in Germany much more pleasant, as they became friends as well as colleagues. Thanks go to Alexander Buchmann, Anja Jerichow, Csaba Korenyi, Isolde Wildgruber, and, once again, to Luca Compagna and Dr. Jorge Cuellar.

I would especially like to thank Warren Toomey for proof-reading my thesis, pointing out a plethora of typographical errors, and for his encouraging feedback.

Thanks also to the Lyx developers for making a great tool. I wrote this whole document using Lyx and had no problems whatsoever. Also thanks to Daniel Jarrot for his LATEX thesis class and Lyx environment. The one which was used to write this dissertation is based on his UQ thesis class.

My family has been very supportive during my studies. In particular, I would like to thank my parents, who provided the financial support which allowed me to continue studying and also to travel to Munich for my internship.

Finally I would like to thank my wonderful girlfriend; Yoonjin Song. Her support and love are greatly appreciated, not to mention her typing assistance. Thank you.

Contents

List of Figures

Chapter 1

Introduction

1.1 Security Protocols

The Internet is a communication platform. It is an architecture upon which a myriad of applications are built. These applications are becoming more complex, more interconnected, and more security sensitive. Gone are the days when the Internet was merely a toy. Consider the volume of corporate communication via email, the increasing prevalence of voice-over-IP telephony, the importance of commercial websites for advertising and the government's increasing reliance on the Internet for information dissemination. Now consider more modern applications like electronic banking, e-commerce, online tax returns and even electronic voting. These applications have strong security requirements which must be guaranteed. An important application such as electronic voting simply cannot tolerate vulnerabilities.

Security protocols (or cryptographic protocols) describe the way in which distributed applications communicate. They define the formats of messages, as well as the way in which messages are exchanged between the involved parties. There are many different aspects of application security, but in the case of a distributed application the security protocol it relies upon is one of the most crucial. This is because the Internet is an inherently insecure medium. Security properties must be crafted on top of this medium using cryptographic techniques.

Vendors of distributed applications either develop their own communication protocols or base their application on protocols which have been standardised by an industrial standardisation group. In either case it can be extremely costly to repair software which has been deployed if a vulnerability is discovered in an underlying security protocol. Security protocols should therefore be designed with care and subjected to scrutiny by experts during their design. Unfortunately for vendors, the design of security protocols is notoriously error-prone, and despite copious public scrutiny, flaws often go undiscovered for many years. The most famous example of this is the Needham-Schroeder Public Key Protocol [30], which contained a critical flaw that went undetected for many years before formal methods were applied to the problem.

A security protocol is a protocol which provides a security property. The simplest and most common examples of security protocols provide authentication and secrecy properties. For example, the Needham-Scroeder Public Key Authentication Protocol [30] provides an authentication property. The term security protocol is becoming broader because security is now a key consideration in the design of many protocols, particularly e-commerce protocols.

Figure 1.1: Alice-Bob Notation

```
A -> B: M1
B -> A: {M2}KeyA
```

Unfortunately, there have been a number of cases where security protocols have been flawed and have not provided the security property claimed by their designers. Sometimes these flaws go undetected for a long duration of time. Security protocols are difficult to design correctly due to the asymmetric and concurrent nature of distributed applications. Flaws are sometimes very subtle and rely on unexpected interactions between concurrent sessions.

When describing security protocols, there is a common notation which is used to show the flow of messages between participants. It is not perfect, but is an easy way to give a description of a flow of messages and can be understood almost immediately. This notation is called Alice-Bob notation. Figure 1.1 shows a typical example of Alice-Bob notation.

The first message of the protocol is sent from A to B, and is M1. The second message of the protocol is sent from B to A and is M2 encrypted with KeyA. There are a number of different styles used to denote encryption, but this thesis will exclusively use the curly-brace style shown above.

The Clark Jacob Library of security protocols contains an introduction to the field of security protocols [18]. It is highly recommended that a reader unfamiliar with the concepts behind security protocols reads this introduction to the topic before reading beyond this chapter.

1.2 Formal Analysis of Security Protocols

Formal analysis has been used with some success to verify the correctness of security protocols. A specification language is used to describe the protocol and its security requirements, and a model checker is then used to verify that the security requirements of the protocol are met. This approach is in some ways superior to human examination because it has the advantage of an exhaustive search through all possible ways in which the protocol and the intruder can behave, and all the ways in which concurrent sessions of a protocol can interact and interfere with each other.

A number of different formal techniques and logics have been applied to the domain of formal security protocol analysis. Most commonly, state space exploration techniques have been used to explore all possible paths through a state space defined by a model of the protocol under analysis. Some other techniques which have been applied to the problem include the Burrows, Abadi and Needham (BAN) logic [12], which is based on the beliefs of principals and inference rules related to these beliefs, inductive theorem proving techniques [42], and the graph-theoretic strand space model [23]. [37] provides a good summary of the current state of formal analysis of security protocols.

The tools used for analysing security protocols are quite varied. Some general purpose tools for describing concurrent systems have been adapted to the purpose, while there are also a number of tools specialised to the field of protocol analysis.

Formal security protocol specification languages have previously been limited in a number of ways. Expressiveness, scalability and ease of use have all been identified as limitations of the current generation of tools. This is changing as new analysis techniques and specification languages are developed.

The specification and analysis of very large complex protocols has been troublesome using the tools currently available. Many of them are based on analysing the Clark-Jacob library [18] of security protocols. This library contains descriptions of a large number of protocols to illustrate their functionality and faults; unfortunately most of the protocols are simple examples and not as complex as the protocols currently being developed by and for industry. The current generation of tools being developed attempts to go beyond the Clark-Jacob library to analyse larger more complex security protocols.

1.3 Limitations of the Currently Available Tools

The process of specifying a protocol can be a tedious, difficult and error-prone task. Improvements in model checking techniques over recent years have made the formal analysis of industrial security protocols more feasible. Advances such as partial order reduction, lazy evaluation, and symbolic model checking [36, 10] have given model checking tools the power to analyse extremely complex specifications quickly and efficiently. Modelling such protocols, however, is not so easy. Languages such as TLA [29] are very general in terms of expressiveness, but require a high level of expertise when it comes to modelling complex security protocols. This is primarily due to their general and low-level nature.

The development of languages specialised to the analysis of security protocols has provided modellers with standard constructs with which to specify things like messages, keys, encryption, and agents. These languages also enforce *restrictions* on modellers. Because they are specialised they must provide support for everything a modeller will need. If this support is not provided, there is usually no simple way to model the required functionality.

An example of this is arithmetic in the High Level Protocol Specification Language (HLPSL) [16, 5]. Arithmetic in HLPSL is difficult because natural numbers are supported, but only as symbols. They have no meaning beyond an identifier for a constant. In order to model functionality similar to counting, a function can be used to refer to the successor of a number, e.g, `SUCC(1)` could refer to two, and `SUCC(SUCC(1))` could refer to three. This approach is limited however, because functions in HLPSL are one-way, and it is difficult to express subtraction. This simple example shows how high level languages can be limiting to protocol modellers.

As the designers of high level languages strive to provide support for the features required by protocol modellers, they must be cautious of the complexity a new feature might add to their language and to the formal models generated by their specifications. It is pointless to add a new feature if the use of this new feature produces a model which the appropriate model checker cannot handle.

The latest generation of protocol specification languages and model-checkers claim to be able to go beyond the Clark-Jacob library of protocols to analyse more of the protocols being used and developed by the industry. This is true to an extent; however they still suffer from some critical limitations. Both HLPSL and muCAPSL [39] have restricted modellers to a single intruder model; the Dolev-Yao intruder model. This is the most powerful intruder model, and is generally the type of intruder which protocols are designed to be secure against. However communication mediums are becoming more diverse and this model is no longer suitable in all cases. Two examples of when this intruder model

is unsuitable are wireless communication, where the intruder may not necessarily be able to block messages, and confidential communication between, for instance, a token and a token reader which must be in contact to communicate.

The restriction to a single intruder model has provided modellers with a simple way to model communication without having to specify all the actions the intruder might take. As security protocols diversify, however, this restriction becomes limiting to protocol modellers.

The analysis of industrial protocols which make use of sub-protocols can be a tedious task. It involves manually combining several protocol specifications into one by modelling the message exchanges of the sub-protocols in the appropriate places in the main protocol. When a protocol makes use of a number of sub-protocols, the specifications can become quite complex and there is a high chance of errors occurring in the specification. Sometimes a protocol will make use of sub-protocols, but not actually specify the protocols to use. In this case modellers (and vendors) must choose an appropriate sub-protocol to model (or build) based on the assumptions the main protocol makes about the sub-protocol. The modelling process can be simplified by a mechanism with which to model the assumptions a protocol makes about a sub-protocol, instead of actually modelling the sub-protocol.

Specifying assumptions about sub-protocols is similar to specifying an intruder model. Both processes affect the way in which messages are sent and received. This dissertation proposes a solution which enables modellers to make use of alternative intruder models and to model a sub-protocol as an assumption or a set of assumptions, rather than modelling the whole protocol.

1.4 Modelling Large Protocols

There is a large body of existing security protocols which provide well defined security properties to applications. Recently, new protocols have been developed which make use of existing protocols as sub-protocols. This makes sense as there are some security properties which are required by many different applications. It is sensible to make use of existing code and designs, rather than creating new protocols which accomplish the same thing as existing protocols. An example of this is mutual authentication. The designers of a large protocol which needs to authenticate two parties to each other may decide to leverage an existing protocol, and use it within their protocol as a sub-protocol.

Unfortunately, the increasing complexity and use of sub-protocols within larger protocols makes it more and more difficult to model security protocols. Once a specification reaches a certain level of complexity it becomes too difficult to comprehend at one time and the chance of an error increases. As part of this research, extensive experimentation has been done using the state-of-the-art protocol specification language HLPSL to model large and complex security protocols. The goal of these experiments was to determine how capable the currently available tools were of modelling large, complex protocols.

The experiments discovered that the HLPSL language was capable of expressing most authentication and key exchange protocols, but also identified a number of limitations of the HLPSL language.

The HLPSL language only supported a single model of the intruder's capabilities. This is limiting as more and more protocols are being proposed for operating in different environments and over new types of media. It should be possible to analyse these protocols, and even just parts of these protocols, with respect to different models of an intruder's capabilities.

In addition to this limitation, the HLPSL language could not be used to model non-repudiation protocols or security protocols with non-repudiation requirements. The language provides no way to specify which actions generate proof evidence, and thus there is no way to reason about an agent's ability to prove or deny its actions or the actions of other agents. Non-repudiation properties are an important part of e-commerce protocols and exchange protocols and this lack of support significantly reduces the applicability of the AVISPA tools.

It was also discovered that modelling large protocols that made use of sub-protocols was difficult and time-consuming. Unfortunately there was no simple way to merge a pre-existing specification of a protocol into a larger specification as a sub-protocol. The specifications had to be manually combined into a single specification. This process was difficult and resulted in unnecessarily complex specifications. These issues were identified as key limitations of the HLPSL language which inhibited the scope of the AVISPA tools.

Note that these limitations are not flaws in the language, but characteristics of the language which make it difficult to specify certain types of protocols. The HLPSL language has a number of limitations other than those outlined above. Some examples are lack of support for arithmetic, fairness constraints, and timestamps. The key limitations of the language which are focussed on by this thesis, however, are the ability to model alternative intruder models, the ability to specify the requirements of non-repudiation protocols, and the difficulty of specifying large, complex protocols in HLPSL. These limitations are highlighted because they can all be solved using a single extension to the language.

Abstract communication channels are a modelling technique which can be used to describe properties of a communication channel. They are used to model properties of a communication medium which are beyond the scope of the analysis. The typical example is a Dolev-Yao channel [21], which models communication over an insecure medium. The channel allows the modeller to specify properties of the communication over this medium. These properties are at a lower level of abstraction than the protocol specification and are thus abstractly expressed using a channel type. Abstract communication channels can be used to abstractly model properties of any part of a protocol which is considered beyond the scope of the analysis. This is not restricted to properties of the physical layer of a protocol. For example, a channel might provide the privacy security property to model a secure channel based on SSL or a similar protocol. Taking this approach further, abstract communication channels can be used when modelling larger application protocols which make use of several sub-protocols. The sub-protocols can be modelled as communication channels which provide certain security properties. Perhaps the high level protocol does not specify the sub-protocol to be used, in which case the properties of the channel must be extracted from the requirements placed on the sub-protocol by the high level application protocol.

1.5 Goals of this Research

This research aims to investigate the effectiveness of the current generation of protocol analysis tools for analysing industrial protocols. Furthermore this research introduces abstract communication channels and experiments with using them to model protocols using a top-down approach. Ultimately, the efficacy of this new concept will be evaluated to determine whether or not abstract communication channels should be integrated into the current generation of protocol analysis tools.

1.6 Organisation of this Thesis

This thesis will be broken up into a number of chapters. Chapter 2 is a review of the literature related to the analysis of security protocols. It provides preliminary information necessary for understanding the rest of the thesis. Chapter 3 presents a new modelling construct; abstract communication channels. Chapter 4 describes three case studies performed to determine the applicability and usefulness of abstract communication channels. Chapter 5 discusses the results of the experiments and future directions related to this work.

Chapter 2

Literature Review

2.1 Modelling Security Protocols

Protocol specification languages have evolved from low-level generic languages such as TLT [20], TLA [29], and PROMELA [26]. These languages are not specialised to security protocols, but can be applied to many types of concurrent systems. They require the modeller to explicitly specify the behaviour of channels, encryption, message composition and decomposition, and many other things related to security protocols. Due to their generality they can be applied to any protocol, but they are unsuitable for general use because of the time it takes to specify protocols, and also because they are not optimised to handle protocol analysis. This creates complex models with enormous numbers of states, and makes the analysis of large protocols with these tools difficult.

A number of different formal techniques and logics have been applied specifically to the domain of formal security protocol analysis, and tools have been developed based on these logics. State space exploration is a technique which can be used to explore all possible paths through a state space defined by a model of the protocol under analysis. Another technique which has been applied to the analysis of security protocols is the Burrows, Abadi and Needham (BAN) logic [12]. This logic is based on the beliefs of principals and inference rules related to these beliefs, for example, if a principal A believes that only B and itself know of a shared key K, and A receives a message encyrpted with K, then A will believe the message was actually from B. BAN logic provides a very high-level view of a protocol. It has been used successfully to identify a number of attacks, however, [41] claims that the BAN logic is flawed and discusses two protocols which, when modelled correctly in BAN logic, were incorrectly found to be secure. [47] counter-claims that the two protocols were modelled incorrectly, and that the BAN logic is not flawed.

Inductive theorom proving techniques [42, 44] are based on sets of rules for extending sequences of events. These rules represent the actions of both honest participants and of the intruder. Authentication and secrecy goals are then expressed as properties of these sequences, and are proved using induction. The general purpose theorom prover Isabelle [43] is used to make the theorom proving process more automated, but these tools are still interactive and require a high level of expertise. Strand spaces [23] are a graph-theoretic approach to representing security protocols. They are closely related to inductive theorem proving techniques, but provide a simpler, more intuitive model and more precise results.

In recent years new specification languages have been developed which are specialised to the domain of security protocols. They provide constructs which allow modellers to easily specify things like

messages, send and receive actions, encryption, and a variety of other capabilities. Some examples of these languages are: HLPSL1.0 [3], CAPSL [40], and Casper [31].

HLPSL1.0 and CAPSL are both languages based on the Alice-Bob notation described earlier. Both languages are based on the idea of a high-level language which is translated into a lower level language for analysis by a number of tools. CAPSL is designed to be a common specification language which can be used by a number of tools. Unfortunately, the language is limited in some ways. For example, it cannot express a situation where a principal receives a message which it cannot immediately decrypt, and it is restricted to secrecy and authentication goals. The HLPSL1.0 language is limited in that it cannot express branching. The Casper approach does not support non-atomic keys. More importantly, Casper is geared towards finite state model checking, and requires restrictive assumptions to be made about the system. For example, the maximum depth of messages must be specified.

These languages are advantageous because they simplify the specification process, and also because the size of the models they generate is considerably smaller than those based on generic languages because things such as send and receive actions are built into the tool and are handled in efficient ways. This approach makes more sense than requiring the modeller to specify things over and over again. These languages are simple, but have expressive limitations. Recently the trend has been to move towards more complicated, yet expressive languages. [37] provides a good summary of the current state of formal analysis of security protocols.

MuCAPSL [38, 39], HLPSL2.0 [5, 16] evolved from languages based on Alice-Bob notation. They both have expressive power beyond their predecessors and aim to increase the scope of security protocol analysis tools to more complex, non-standard protocols. This approach is a reflection of the increasing diversity of computer communication applications, and hence the increasing diversity of security protocols themselves. Both HLPSL2.0 and MuCAPSL are based on separate specifications of roles, which are interpreted as independent state machines. MuCAPSL is specifically targeted towards multicast and group protocols, whereas HLPSL2.0 aims to be general purpose and flexible. This thesis will refer to HLPSL2.0 as HLPSL from now on.

2.2 The Dolev-Yao Intruder

The Dolev-Yao intruder model [21] is the strongest possible model of the capabilities of an intruder which preserves the properties of encryption. The model states that the intruder can read all messages which are sent, block any message which is sent, and arbitrarily re-direct messages. The intruder can store messages it receives indefinitely, and can arbitrarily re-order messages. Furthermore, the intruder has the capability to decompose messages into their components and to compose messages from other pieces of its knowledge. The intruder can encrypt and decrypt messages if it possesses the appropriate key.

The only restriction that is placed on the Dolev-Yao intruder is that it cannot break encryption. If it receives an encrypted message, it cannot learn the contents of the message unless it has knowledge of the appropriate key.

2.3 Fair Exchange Protocols

Fair exchange protocols are an example of a class of protocol that the tools currently available have difficulty analysing. A number of distributed applications involve an electronic exchange of some description. Data is exchanged between parties. In situations where the data is valuable it is desirable for all honest parties to ensure that the exchange is atomic. Fair exchange refers to an exchange of digital items which allows no participant to gain an advantage over another. This principle has been applied to several applications.

Electronic contract signing is an application of fair exchange which allows parties to exchange signatures on a pre-agreed text: the contract. Fair electronic contract signing ensures that no party can obtain a signed contract from another without also signing a copy of the same contract and making it available to the other participant(s). Without this guarantee it might be possible for malicious parties to manipulate contract signing procedures and avoid signing the corresponding contract.

E-commerce applications also participate in electronic exchanges. Usually electronic payment is exchanged for a receipt, and eventually delivery of a product. However it is becoming increasingly common to purchase electronic items on-line, for example, digital music from an online store. In this situation the exchange in completely electronic. The customer provides payment authorisation, and the merchant provides the digital item. A guaranteed fair exchange will protect both parties from the other. Consumers can be sure that they will receive the goods they pay for, and merchants can be sure they will be paid. This reduces the level of trust required between both parties, and hopefully can increase consumer acceptance of electronic payment systems.

Another application of fair exchange is certified email. Certified email ensures that the recipient cannot receive an email without also providing proof that he has received it. This can make it difficult to deny receiving an email. Although certified email is not as far-reaching as the other examples, it illustrates how fair exchange can be applied to any domain which involves a digital exchange.

Fair exchange protocols, including certified email and electronic contract signing protocols, are designed to meet requirements which are different from typical secure communication protocols. The broad goal of all fair exchange protocols is to ensure that no party can take advantage of another. Stemming from this goal are the concrete requirements of each protocol. Unfortunately, these requirements are not standardised and differ from protocol to protocol. The tools currently available have trouble analysing the requirements of fair exchange protocols and a number of experiments have been done which apply the currently available tools to these protocols [25, 46, 24, 27, 13, 14, 45]. These experiments have had considerable success verifying the fairness properties of fair exchange protocols, but have so far not attempted to analyse the non-repudiation properties of these protocols. This analysis is, unfortunately, beyond the capabilities of the currently available tools.

2.4 The AVISPA Project

The AVISPA project is a European project which is developing a state-of-the-art set of tools for protocols analysis. The website for the AVISPA project (www.avispa-project.org) states that:

> AVISPA aims at developing a push-button, industrial-strength technology for the analysis of large-scale Internet security-sensitive protocols and applications. This tech-

nology will speed up the development of the next generation of network protocols, improve their security, and therefore increase the public acceptance of advanced, distributed IT applications based on them.

We at AVISPA will achieve this by advancing specification and deduction technology to the point where industry protocols can be specified and automatically analysed.A central aim of the project is then to integrate this technology into a robust automated tool, tuned on practical, large-scale problems, and migrated to standardisation bodies, whose protocol designers are in dire need of such tools.

AVISPA is a shared-cost RTD (FET open) project, funded by the European Commission under the Information Society Technologies Programme operating within the Fifth Framework Programme, started on January 1st, 2003.

2.4.1 The AVISPA tools

The AVISPA tools consist of a translator from the High Level Specification Language (HLSPL) into the Intermediate Format (IF), and four back-ends with which to analyse the generated IF specification. Each of the four back-ends may make use of a further translator in order to convert the IF file into the tool's individual specification language. The four back-ends are the On-the Fly Model Checker (OFMC), SAT based model checker (SATMC), Constraint Logic Attack Searcher (CL-ATSE) and Tree Automata based automatic approximations for the analysis of Security Protocols (TA4SP). All four back-ends must be able to parse the generic IF file and must comply with a standard output format. This makes automated test runs of all four tools over a large number of specifications simpler, and will soon allow a graphical front-end to parse results for visual display.

2.4.2 The High Level Protocol Specification Language (HLSPL)

HLPSL [5, 17] is the protocol specification language of the AVISPA project. It was designed to be a fast, easy to use protocol specification language which would be accessible to protocol engineers who might not necessarily be well versed in formal methods.

HLPSL is based on some of the concepts of TLA [29]. The semantics of HLPSL are wholly defined using TLA. HLPSL provides a flexible, theoretically sound protocol specification language which is sufficiently high level to be accessible, yet expressive enough to be able to model most security protocols.

The HLPSL language supports branching, non-determinism, multiple layers of encryption, functions, sets, role composition, and even allows the intruder to participate in protocol sessions as a legitimate player.

Each HLSPL specification is made up of basic roles and compositional roles. Basic roles define the initial knowledge and the behaviour of each of the participants. Compositional roles are used to instantiate the roles with values and to define protocol sessions.

Each basic role contains a parameter list which describes the initial knowledge it must be instantiated with. A basic role also has a `played_by` parameter, which is used to instantiate the role with a player. Each basic role also contains a list of local variables and an initialisation section, and finally, a list of transitions. A basic role is used to define the behaviour of an honest participant. This is done using a list of transitions. Each transition has a left hand side which describes what must be true for the

Figure 2.1: Architecture of the AVISPA tool-set

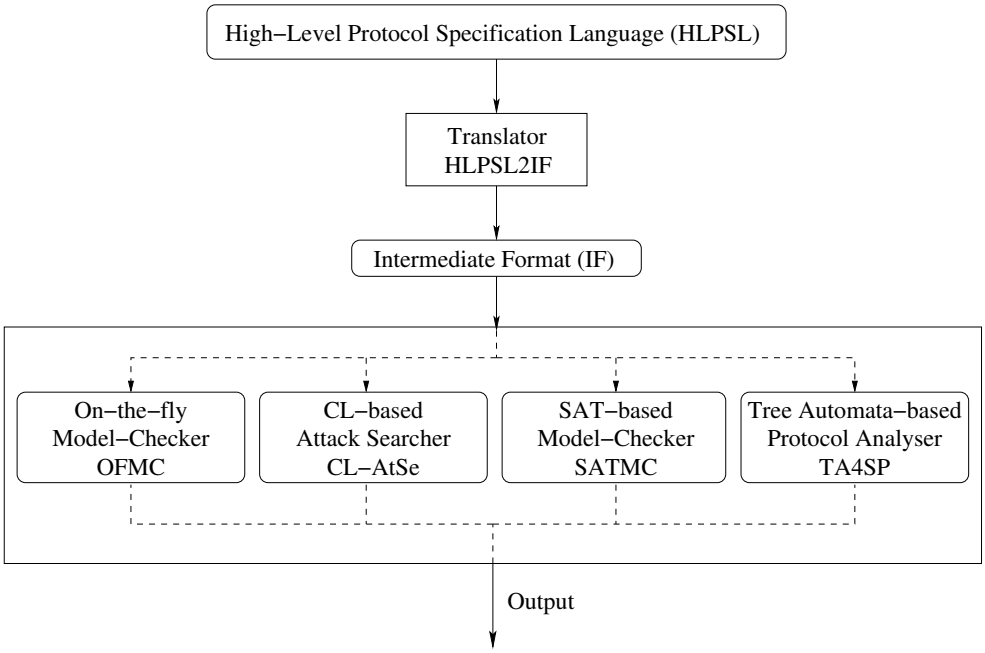


Figure 2.2: A HLSPL transition

```
1. State  = 0 /\ RCV(Text) =|>
    State' = 1 /\ SND(Text)
```

transition to be enabled, and a right hand side which defines the consequences of that transition being fired. An example of a HLPSL transition is in Figure 2.2.

In this example the transition is fired if the variable State is equal to 0 *and* the message Text is received on the channel RCV. This transition is only triggered when the message received on the channel RCV is equal to the value of the variable Text. If the transition is fired, the value of State is changed to 1. The syntax State' (spoken as "state-prime") refers to the value of State immediately after the transition is complete. This syntax for referring to the new value of a variable is taken from TLA. As well as updating the value of State, a message containing the value of Text is sent on a channel called SND.

Note that the variables State, RCV, SND, and Text all need to be declared and given appropriate types. The different types available in HLPSL are described in [5], but are not discussed here in depth. They include support for messages, agents, keys, nonces, natural nonces, and Dolev-Yao channels.

Figure 2.3 is a more advanced example of a HLPSL transition. It demonstrates concatenation, encryption, nonce generation, and the binding of values received in messages to variables. Concatenation is

Figure 2.3: A more advanced HLSPL transition

```
1. State  = 1 /\ RCV({Text.Val'}KeyA) =|>
    State' = 2 /\ SND({Text.Nonce'}KeyA)
```

Figure 2.4: A HLSPL compositional role

```
role Session (
    A,B : agent,
    KeyAB : symmetric_key,
    SND,RCV : channel (dy)
) def=

  composition

    Alice(A,B,KeyAB,SND,RCV)
 /\ Bob(A,B,KeyAB,SND,RCV)

end role
```

expressed using the "." operator. In this example the values of Text and Nonce are being concatenated together for the SND action. Encryption is expressed using curly braces, i.e. "{" and "}", followed by the key to be used for encryption. In the send action of Figure 2.3 the values of Text and Nonce are concatenated together then encrypted with the key KeyA. Nonce generation refers to the creation of a new value which has not been used before. If a variable is of type `text (fresh)`, then it can be used as a nonce. The example shows the new value of Nonce being sent on the channel SND.

Receive actions can be used to both restrict the messages which will be accepted, and to bind the values received to variables. For example, the receive action in the example will only accept messages which are the value of Text concatenated with some other message, all encrypted with KeyA. After this transition is completed, Val will be equal to the value of the component of the message it corresponds to.

Compositional roles in HLPSL are used to instantiate the basic roles, or other compositional roles. They declare variables, and pass them to the roles they instantiate. This is how shared initial knowledge is specified. In addition to this, compositional roles are used to define the initial knowledge of the intruder, and to specify which of the roles the intruder might be able to play legitimately. Figure 2.4 shows a typical compositional role named Session. Session takes a number of arguments, which it passes onto the basic roles Alice and Bob when it instantiates them. The "/\" operator in a composition section indicates parallel composition of the roles. That is, they will both be executed simultaneously as independent state-machines.

Figure 2.5: A HLSPL top level compositional role

```
role Environment () def=

  const
    a,b : agent,
    keyAB,keyiB : symmetric_key,
    snd,rcv : channel (dy)

  knowledge(i) = {a,b,keyiB}

  composition

    Session(a,b,keyAB,snd,rcv)
 /\ Session(i,b,keyiB,snd,rcv)

end role
```

Figure 2.6: HLPSL goals

```
goal
  Bob authenticates Alice on Msg
  Secrecy of Msg
end goal
```

Figure 2.5 shows another compositional role called Environment. This is the top-level role which instantiates the Session role. Notice that it instantiates two Session roles, one between the honest agents a and b, and another between the intruder (i), and b. This means that intruder has the ability to play as a legitimate Alice, and has a pre-existing relationship with b, and even an established key called keyiB.

The HLPSL language has been designed to support temporal logic style security goals; however the AVISPA tools do not yet provide this support. Goals are currently specified as macros. There are three types of goal macros available: secrecy, strong authentication, and weak authentication. These three goals can be used to capture the requirements of many protocols, but are not sufficient to model many others. The specification of goals is HLPSL is done in the goal section. Figure 2.6 is an example of two HLPSL goals.

This section has described HLPSL in a manner which provides the reader with a basic understanding of the way in which HLPSL works, without going into a great deal of depth.

The next section discusses IF, the lower level language that HLPSL specifications are translated into.

Figure 2.7: The type symbols section of the standard IF prelude file

```
section typeSymbols:
  agent, text, symmetric_key, public_key, function, table,
  message, fact, nat, protocol_id
```

2.4.3 The Intermediate Format (IF)

The IF [6] language is a low-level tool-independent protocol specification language based on term-rewriting. IF specifications describe a transition system in terms of an initial state, and a list of rules which dictate the ways in which this state can change. IF specifications are broken up into two files. The first file is called the prelude file, and is used to describe symbols, rules and equations which are protocol independent. There is a standard IF prelude file which is used for all protocol specifications. The IF prelude file itself is broken up into a number of sections: the type symbols section, the signature section, the types section, the equations section, and the intruder section. The Type Symbols section lists all of the types which will be used throughout the rest of the prelude file and the protocol specific IF specificaitons. Figure 2.7 shows the type symbols section of the standard IF prelude file. It defines all the types which are available for IF specifications.

The signature section of the IF prelude file performs two tasks. Firstly, it defines the type heirarchy for the types which are listed. For instance, it specifies that text is a sub-type of type message. In addition to this, the signature section also contains type signatures for all the facts which will be used in IF specificaitons. Figure 2.8 shows the signature section of the standard IF prelude file.

The signatures displayed in the second part of Figure 2.8 show all of the standard IF functions and facts which are used for representing and reasoning about security protocols. For example, the first entry describes the pair function, which has two parameters of type message, and which returns a message. The other functions are defined in the same manner. Pair is used for expressing tuples, crypt is used for public key encryption, inv is used for the inverse function, scrypt is used for symmetric encryption, exp is for exponentiation, xor is for exclusive or, apply is used for expressing the application of functions, iknows is for knowledge of the intruder, contains is used for sets, and witness request, wrequest and secret are used for augmenting specifications with facts used when expressing security requirements. The signature for each of these facts specifies its arity, the types of its parameters, and the type of the result of this function or fact.

The types section of the prelude file declares a number of symbols and the type of each of the symbols. The equations section then uses these symbols to define algebraic properties of the functions defined in the signatures section. Figure 2.9 shows the types and equations sections of the standard prelude file. It defines the associativity of the pair functions, the self-inverse property of the inv function, the commutativity of the exp function, and the associative, commutative and self-inverse properties of the xor function. These properties are defined in terms of the symbols declared in the types section. For example, `inv(inv(PreludeM)) = PreludeM` means that the inverse of the inverse of any message is equal to that message.

The final section of the prelude file is the intruder section, which defines the capabilities of the intruder in terms of the composition and decomposition of messages, and the generation of new fresh values.

Figure 2.8: The signature section of the standard IF prelude file

```
section signature:
  message > agent
  message > nonce
  message > symmetric_key
  message > public_key
  message > function
  message > table
  message > set
  pair     : message * message -> message
  crypt    : message * message -> message
  inv      : message -> message
  scrypt   : message * message -> message
  exp      : message * message -> message
  xor      : message * message -> message
  apply    : message * message -> message
  iknows   : message -> fact
  contains : message * message -> fact
  witness  : agent * agent * protocol_id * message  -> fact
  request  : agent * agent * protocol_id * message * nat -> fact
  wrequest : agent * agent * protocol_id * message * nat -> fact
  secret   : message * agent -> fact
```

Figure 2.9: The types and equations sections of the standard IF prelude file

```
section types:
  PreludeK,PreludeM,PreludeM1,PreludeM2,PreludeM3 : message
section equations
  pair(PreludeM1,pair(PreludeM2,PreludeM3)) = pair(pair(PreludeM1,PreludeM2),PreludeM3)
  inv(inv(PreludeM)) = PreludeM
  exp(exp(PreludeM1,PreludeM2),PreludeM3) = exp(exp(PreludeM1,PreludeM2),PreludeM3)
  exp(exp(PreludeM1,PreludeM2),inv(PreludeM2)) = PreludeM1
  xor(PreludeM1,xor(PreludeM2,PreludeM3)) = xor(xor(PreludeM1,PreludeM2),PreludeM3)
  xor(PreludeM1,PreludeM2) = xor(PreludeM2,PreludeM1)
  xor(xor(PreludeM1,PreludeM1),PreludeM2) = PreludeM2
```

Figure 2.10: The intruder section of the standard IF prelude file

```
section intruder:
  % generate rules
  step gen_pair (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(pair(PreludeM1,PreludeM2))
  step gen_crypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(crypt(PreludeM1,PreludeM2))
  step gen_scrypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(scrypt(PreludeM1,PreludeM2))
  step gen_exp (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(exp(PreludeM1,PreludeM2))
  step gen_xor (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(xor(PreludeM1,PreludeM2))
  step gen_apply (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(apply(PreludeM1,PreludeM2))
  % analysis rules

  step ana_pair (PreludeM1,PreludeM2) :=
    iknows(pair(PreludeM1,PreludeM2)) => iknows(PreludeM1).iknows(PreludeM2)
  step ana_crypt (PreludeK,PreludeM) :=
    iknows(crypt(PreludeK,PreludeM)).iknows(inv(PreludeK)) => iknows(PreludeM)
  step ana_scrypt (PreludeK,PreludeM) :=
    iknows(scrypt(PreludeK,PreludeM)).iknows(PreludeK) => iknows(PreludeM)
  % Generating new constants of any type:
  step generate (PreludeM) :=
    =[exists PreludeM]=> iknows(PreludeM)
```

The knowledge of the intruder is treated in the same way as other facts and functions in IF, using the iknows fact. The equations is Figure 2.10 use the iknows function to reason about the composition and decomposition of messages. This is the final section of the IF prelude file. The capabilities of the intruder are defined in a similar way to the properties of functions. This is because the knowledge of the intruder is desribed using the `iknows` function. For example, the first entry in the intruder section defines the ability of the intruder to generate a pair. If the intruder knows `PreludeM1` and `PreludeM2`, then he also knows `pair(PreludeM1,PreludeM2)`.

Individual IF specifications are generated from HLPSL specifications using the tool hlpsl2if. Each IF specification is broken up into a number of sections in a similar way to the IF prelude file: the signature section, the types section, the inits section, the rules section and the goals section. The signature section of an IF specificaiton is similar to the signature section for the IF prelude file, except that it contains fact signatures whcih are used to describe the state of honest agents. The types section contains type declarations of all the symbols which will be used throughout the specification. These symbols correspond to the variables and constants declared in the HLPSL specification. The inits section is used to describe the intial state of the system. It states the initial knowledge of the intruder,

Figure 2.11: An IF Rule

```
step step_0 (C,S,K_CS,MD5,Hello,Success,Ts,Dummy_Timestamp,Timestamp,SID) :=
 state_Client(C,S,K_CS,MD5,Hello,Success,Ts,0,Dummy_Timestamp,SID).
 iknows(pair(Hello,Timestamp))
 =>
 state_Client(C,S,K_CS,MD5,Hello,Success,Ts,1,Timestamp,SID).
 iknows(pair(C,apply(MD5,pair(Timestamp,K_CS)))).
```

Figure 2.12: An example of an IF goals section

```
section goals:
goal authenticate_Timestamp (C,S,Timestamp,SID) :=
 request(S,C,timestamp,Timestamp,SID) &
 not(witness(C,S,timestamp,Timestamp)) &
 not(equal(C,i))
```

as well as the state of the honest agents. The rules section contains a list of IF rules which correspond to the transitions of the HLPSL specification. Each rule describes how the state of the system can change. Figure 2.11 shows an example of an IF rule. Note that all of the symbols in rules such as Figure 2.11 must be defined and intialised at the beginning of the IF file.

Attack states are defined in the goals of an IF file. Each goal describes a state which must never be reached. Figure 2.12 shows an example of a goals section.

When a model checker processes an IF file, it must search through the state space created by the inital state and the different rules listed in the rules section of the IF file. The model checker searches for attack states which are defined in the goals section of the IF file. If an attack state is reached, the transition path from the intial state to the attack state is displayed. This path describes *how* the attack occurred. The name of the goal which was violated is also displayed. If no attack states are reached, then the model-checker displays a message indicating that no attacks were found.

This chapter has discussed different techniques of formally modelling and analysing security protocols, as well as introduced the reader to AVISPA tools. The next chapter will describe a new technique for specifying large and complex security protocols. This new technique is a modelling construct which allows modellers to specify the assumptions they would like to make about a communication between parties. It has been implemented as an extension to the AVISPA tools, which is why this chapter provided an introduction to the HLSPL and IF languages.

Chapter 3

Modelling Large and Complex Protocols

The previous chapter concluded with a discussion of some of the limitations of the current generation of protocol specification and analysis tools, including the difficulty of modelling complex protocols and the lack of support for alternative intruder models. This chapter present abstract communication channels as a solution to these limitations. Section one explains why abstract communication channels are a suitable solution to the problems discussed in the previous chapter. Section two describes exactly what abstract communication channels are, and the functionality they provide. Section three describes how abstract communication channels have been implemented; including how they are specified, and how the tools implement the required functionality of each channel.

3.1 Why Abstract Communication Channels

Abstract Communication Channels allow protocol modellers to explicitly state assumptions they would like to make about a communication medium. They can be used to specify any property of a communication medium which is below the level of abstraction which the modeller wishes to use; either physical properties of the medium, or properties of a sub-protocol which is being used. Abstract communication channels allow modellers to build large protocols based on the security properties of sub-protocols which they have already verified, or which they assume to be correct. This dissertation presents abstract communication channels as a new protocol modelling technique which both simplifies the specification process and increases the expressiveness of high level protocol specification languages. Abstract communication channels not only allow analysts to specify alternative intruder models, they can also be used to simplify the specification of large protocols which make use of sub-protocols. Additionally, abstract communication channels can be used to reason about non-repudiation properties of security protocols. This section explains why abstract communication channels are an appropriate and useful extension to protocol modelling languages, and why they are a suitable mechanism with which to increase the expressiveness of high level protocol specification languages.

The key advantage to a channel-based approach for specifying assumptions about communication is that channels provide a high level of flexibility. By creating a number of different named channels, each of a different type, it is possible to model situations where some messages are sent using one communication model and others are sent using different models of communication. This capability is extremely important for some of the protocols under development. For example [19] and [8] both

discuss a scenario where a user wishes to confidentially send a document to a public printer from a PDA. A physical connection is used to establish a keyed link between the PDA and the printer, negating the need for a Public Key Infrastructure. This situation is impossible to model without the ability to specify different intruder models for different parts of a protocol (i.e, channels).

Another advantage to the channel approach is extensibility. Each unique set of assumptions made about some communication is represented as a different channel type. This makes it easy to add new intruder models to the language without having to modify the grammar beyond a new keyword with which to describe the new type of channel. The behaviour of this new channel is then described by an appropriate translation of send and receive actions into the lower level term-rewriting system. Therefore, as new intruder models are envisaged, they can be added with ease, and without the need to teach modellers about changes to the language.

The AVISPA tools aim to develop tools which are so fast and easy to use that they can be used for experimentation during the *design* phase of a protocol. Allowing modellers to specify sub-protocols gives protocol designers an easy way to experiment with a large protocol like SET. There is no need to specify how the assumed sub-protocol works or what it is, only the security properties it supplies. This enables a top-down approach to protocol design and analysis, where a protocol designer first specifies their needs, and can then incrementally instantiate a protocol with a concrete realisation of those needs. Alternatively, after specifying the needs of a sub-protocol, the designers may search for a protocol they can use which provides the properties they require, and perhaps even analyse this protocol separately.

The ability to include protocol analysis in the design phase of a protocol will lead to clearer security goals and more secure protocols. Using channels to model sub-protocols makes this more feasible and also allows a top-down approach to the analysis of design.

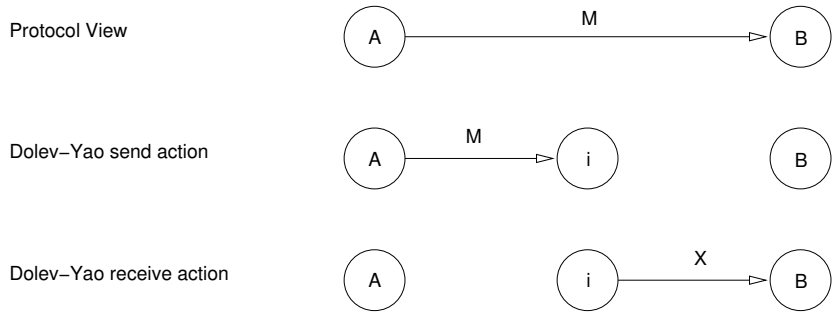
Finally, abstract communication channels provide a convenient way for modellers to compose specifications together. The ability to abstractly specify the properties of a sub-protocol means that modellers can keep specification of a main protocol and its sub-protocol separate. There is no need to include the messages and data of each sub-protocol in the main protocol. Instead, the sub-protocol can be modelled and analysed separately, its relevant security properties verified and extracted and the main protocol can simply us an appropriate channel type. This has the advantage of keeping specifications smaller, which makes them easier to write, easier to understand, and also easier to analyse because of the reduced amount of states. It also means that once a protocol has been modelled and its security properties analysed, there is no need to model it again and again each time it is used as a sub-protocol. The mapping between security goals and an appropriate channel type is a manual task and is restricted to the channel types available. If a protocol provides a property not captured by an existing channel type then it might be useful to add this type of channel to the tool.

Abstract communication channels are a simple, flexible, and extensible construct which allows modellers to specify alternative intruder models, as well as security properties provided by sub-protocols. Both of the capabilities are important to modeller as protocols are developed for more diverse environments and as they begin to rely more and more on sub-protocols.

3.2 Abstract Communication Channels

Abstract communication channels are a modelling construct which allow modellers to make assumptions about a communication medium. For example, if a modeller wants to specify that a message is

Figure 3.1: The behaviour of Dolev-Yao channels



sent privately, the modeller can use an abstract communication channel to send the message.

A channel is a named communication medium. Messages can be sent to channels and received from channels. Exactly what happens when a message is sent or received on a channel is dependent on the type of the channel. This defines the properties of the channel.

The HLPSL language supports channels, however the only channel type available is a Dolev-Yao channel. This channel type is used to specify communication over a medium which is controlled by the Dolev-Yao intruder model. Because the Dolev-Yao intruder model gives the intruder complete control over the network, there has been no need to separate the channel from the intruder. All messages are sent to the intruder, and all messages are received from the intruder. Figure 3.1 illustrates the behaviour of a Dolev-Yao channel.

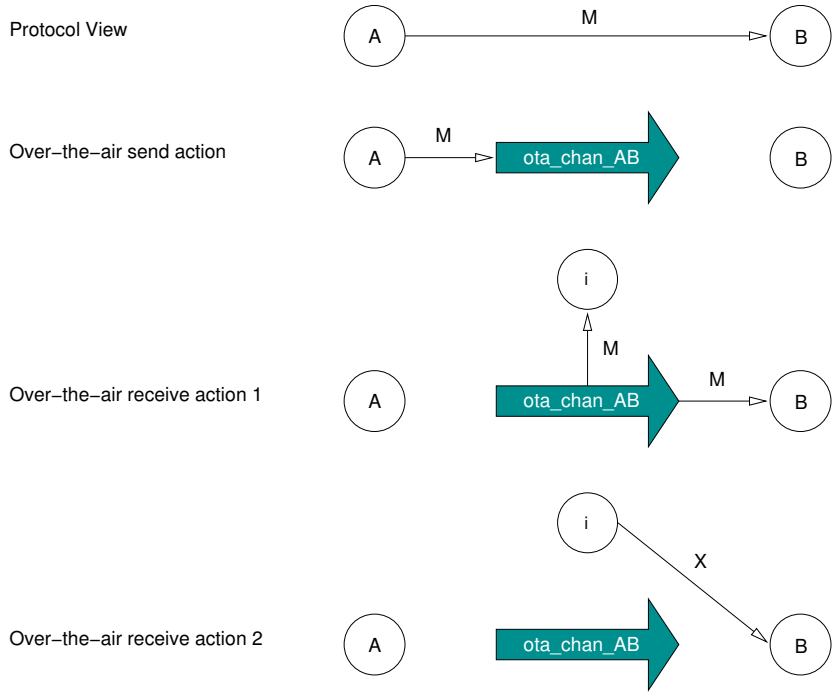
In the receive action of a Dolev-Yao channel the intruder sends the value X. This indicates that the intruder can send any value here, including M. It is important to remember that the value which the intruder sends must be acceptable by the recipient. For example, if the recipient is expecting to receive the name of an agent and a predefined message constant, both encrypted with its own public key, then it will only accept a message of this form. If the intruder is capable of constructing a message of this form, then this message will be X.

Abstract communication channels are channels that behave differently from Dolev-Yao channels. In some cases messages are not sent to the intruder and the channel itself is used to store messages. In other cases receive will be triggered by the channel instead of the intruder. Some channels generate special proof facts which can be used when specifying the channel requirements. A summary of the new types of communication channels is provided in this section.

3.2.1 Named Channels

Prior to the extensions made to the AVISPA tools as part of this research, channel names were superfluous information. Send and receive actions of Dolev-Yao channels never actually sent information to a channel. Abstract communication channels sometimes lead to a situation where agents must have access to the *same* channel in order to communicate over it. This is part of the semantics of abstract communication channels. It should be noted to modellers familiar with the AVISPA tools that they

Figure 3.2: The behaviour of over-the-air channels



will now need to ensure that agents are given access to the same channels if they wish to communicate using the new channel types. This should be done when instantiating the various roles from compositional roles. The process is discussed in Section 3.3.1.

3.2.2 Over-the-air Channels

Over-the-air channels deliver the message to the intruder at the same time as they deliver the message to the recipient. This is used to model a wireless communication medium in which it is difficult for the intruder to intercept messages on-the-fly. When an agent sends a message on an over-the-air channel; it is received by the intruder at the same time as it is received by the intended recipient. It is possible for the intruder to send messages over an over-the-air channel, however the intruder's ability to manipulate a protocol on-the-fly, such as in man-in-the-middle attacks is limited by this communication model. Figure 3.2 illustrates the behaviour of over-the-air channels. Notice that there are two possible receive actions, one in which the message is received from the channel by the recipient and the intruder, and another where the intruder tricks the recipient into accepting another message. The other message, X, could be identical to M, but is named differently because it might be different.

Some models of over-the-air communication stipulate that the intruder cannot block messages, and thus all messages are eventually delivered to the recipient. The IF language cannot support fairness constraints such as "eventually", and so this functionality is not included. Other tools which might make use of abstract communication channels should consider including this in their model of over-

the-air communication. Section 5.3.1 discusses some other channel types which could be supported by tools with the ability to express fairness constraints.

3.2.3 Authentic Channels

A common requirement of a security protocol is authentication. In fact, there is a whole class of security protocols commonly referred to as authentication protocols. In a simple example scenario where Alice sends message to Bob, one can say that Bob wishes to authenticate Alice on message. This means that Bob wants to be absolutely certain that message was sent by Alice.

As well as being a common type of security protocol, authentication protocols are commonly used as sub-protocols for more complex protocols. Therefore an authentic communication channel has been introduced. An authentic channel allows messages to be sent and received on it. When a message is sent over an authentic channel the knowledge of the channel is updated with the message *and* the knowledge of the intruder is updated. When an agent receives a message on an authentic channel, it specifies the expected sender of the message. If an appropriate message hasn't been sent by the specified action then no message can be received.

It is important to remember that all actions in HLPSL are actually *reactions.* They must be triggered by an event, usually a receive event. Once a receive event is triggered, the result usually contains a send action. In order for a reaction to be triggered, it must first be *enabled*. In a situation where an agent is ready to receive a message on an authentic channel, but an appropriate message has not been sent by the expected agent, the receive action is not enabled.

Figure 3.3 shows the behaviour of an authentic channel. Notice that the channel can now receive and send messages instead of just the intruder as with the previous model on communication.

Channels behave in a similar way to the intruder. Messages sent to a channel are remembered forever, and may be sent on at an arbitrary time, or not at all. The model-checking tools will explore every possible way in which message could be delivered and ordered.

There are two commonly used classes of authentication: Strong authentication and weak authentication. The channel presented here provides weak authentication. Strong authentication requires that the message sent by Alice is never received by Bob more than once. This is referred to as replay protection.

3.2.4 Confidential Channels

Confidential channels are another commonly used sub-protocol. The main protocol assumes that the involved parties have established a secure channel with which they can communicate privately. Confidentiality does not ensure authenticity, as it might be possible for the intruder to inject messages into the channel. When sending on a confidential channel, the knowledge of the intruder is not updated whilst the knowledge of the channel is. A receive event on a confidential channel is enabled if an appropriate message has been sent on the channel, *or* if the intruder has knowledge of an appropriate message. It is important to realise that the intruder may be able to construct a message that the receiver will accept, even though he does not receive the messages being sent. Figure 3.4 and illustrates the send and receive actions on a confidential channel.

Figure 3.3: The behaviour of authentic channels

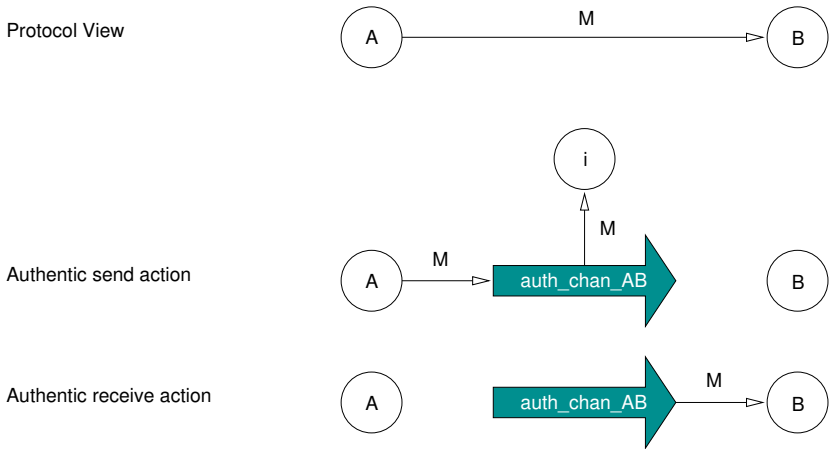


Figure 3.4: The behaviour of confidential channels

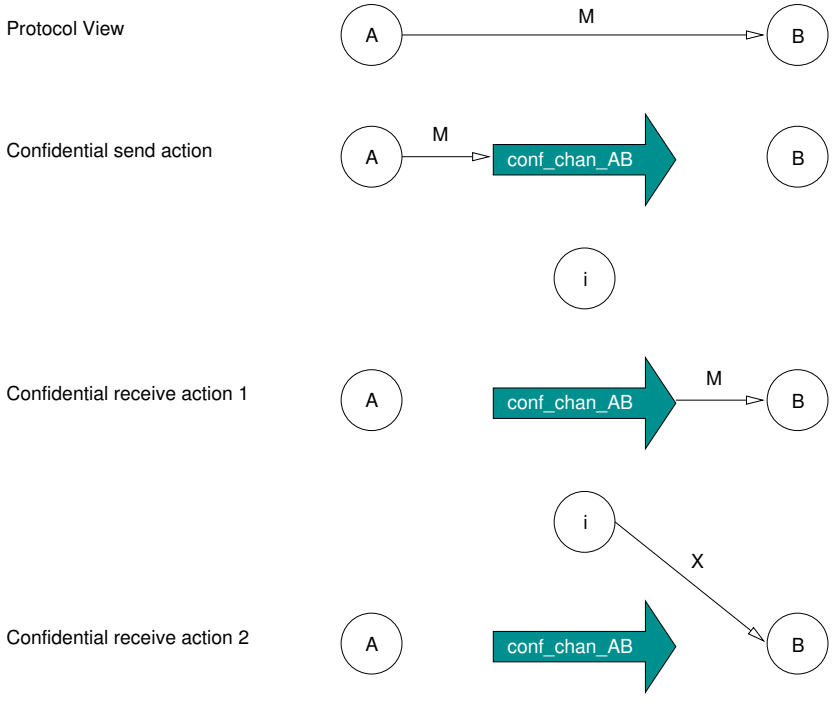
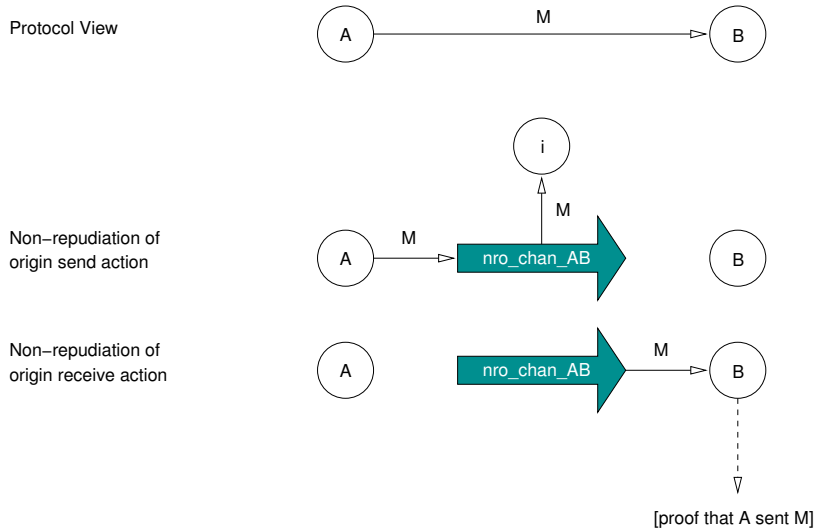


Figure 3.5: The behaviour of a non-repudiation of origin channel



Note that both confidential and authentic channels can be used to model sub-protocols or to model assumptions about physical properties of the communication medium. This applies to Abstract Communication Channels in general.

3.2.5 Non-repudiation of Origin Channels

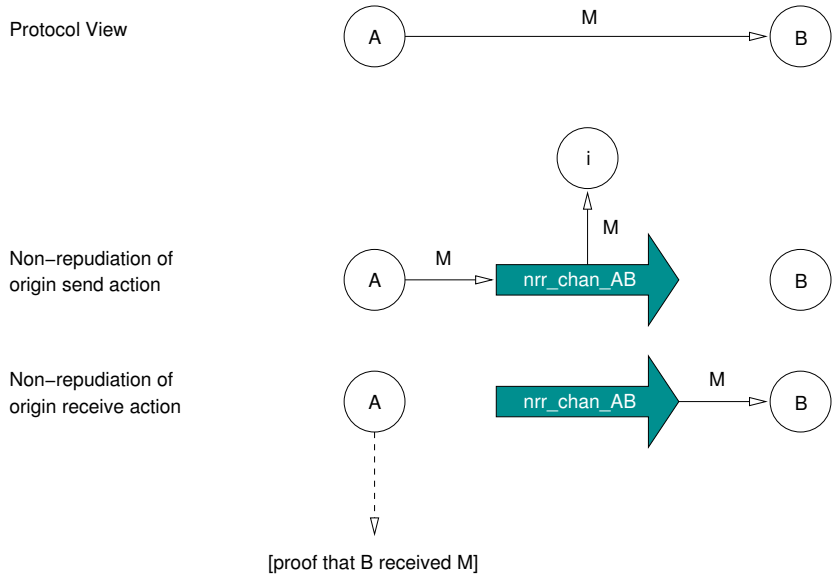
A number of protocols, particularly e-commerce protocols, have requirements involving non-repudiation properties. These properties involve an agent's ability to prove that certain messages were sent or received by other agents. The latest generation of model checking tools do not have support for the specification of non-repudiation security requirements. In fact, this is an area of ongoing research. Abstract communication Channels can be used to indicate which messages generate which types of proofs. Security requirements can then express non-repudiation properties based on the proofs generated by these channels.

The non-repudiation of origin property states that an agent may not deny sending a particular message. In other words, the recipient can prove that the sender sent the message. A non-repudiation of origin channel is identical to an authentication channel except that upon receipt of a message over a non-repudiation of origin channel, the recipient is given the ability to prove that the sender really sent the message. Figure 3.5 illustrates send and receive actions on a non-repudiation of origin channel.

3.2.6 Non-repudiation of Receipt Channels

Non-repudiation of Receipt means that the recipient of a message cannot deny receiving it. Alternatively, the sender of a message can prove that the recipient *did* receive a particular message. Non-repudiation of receipt is usually achieved using a series of messages which make up a non-repudiation

Figure 3.6: The behaviour of a non-repudiation of receipt channel



protocol. However, Abstract Communication Channels allow modellers to specify this property without worrying about the details of the sub-protocol. A non-repudiation of receipt channel is an authentic channel that generates a proof when messages are received on the channel. This proof allows the sender to prove that the recipient did receive the message. Figure 3.6 demonstrates the behaviour of a non-repudiation of receipt channel.

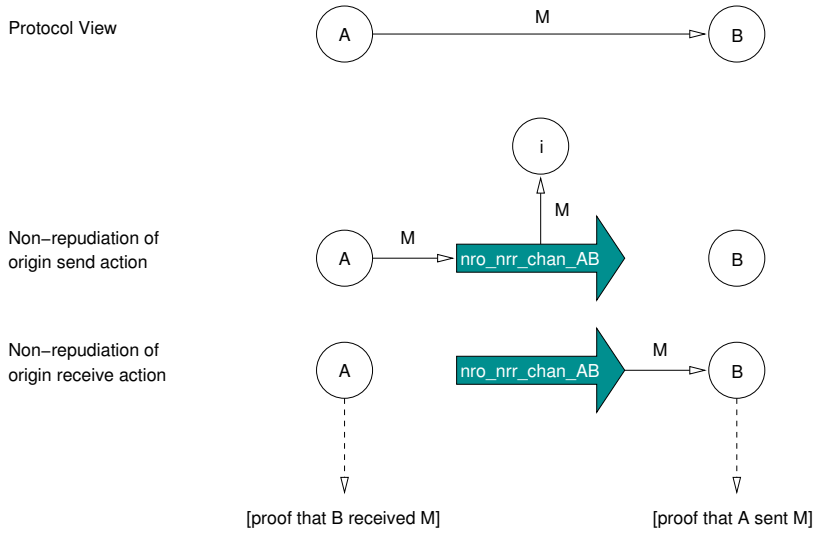
3.2.7 Non-repudiation of Origin and Receipt Channels

Some protocols provide both non-repudiation of origin and non-repudiation of receipt. To allow modellers to easily specify these properties for a sub-protocol, a channel type has been created which provides a proof that the sender sent the message to the recipient, and a proof that the recipient received the message to the sender. Figure 3.7 illustrates send and receive events over non-repudiation of origin and receipt channels.

All channels which provide a non-repudiation property are also authentic channels. This is for the following reason: if an agent is to be provided with proof that a certain message was sent or received by another agent, then the message must have actually been sent or received by that agent. It does not make sense to have a non-repudiation of origin channel that generates proofs that an agent sent a message, if the message wasn't actually sent.

The proofs generated by non-repudiation channels are meaningless on their own. However, they give a modeller the ability to specify security requirements involving proofs. For instance, the modeller may wish to specify that at the conclusion of a protocol, Alice must be able to prove that Bob received a certain message.

Figure 3.7: The behaviour of a non-repudiation of origin and receipt channel



3.3 Implementing Abstract Communication Channels

In order to experiment with the proposed extension, abstract communication channels were added to the AVISPA tools. To implement the functionality described in the previous section a number of changes needed to be made to the tools. The HLPSL specification language was modified to allow modellers to declare abstract communication channels and to express send and receive actions over them, and the hlpsl2if tool which translates HLPSL specifications into IF specifications was modified to support these new constructs. In addition the IF prelude file was extended with new facts used to reason about abstract communication channels.

The scope of the changes made to the tools was deliberately limited to the translation process. The HLPSL language, and the hlpsl2if translator have been extended to support the declaration of the new channel types through the channel attribute construct. They have also been extended to support message parameters, which are used when sending and receiving messages on abstract communication channels. Finally, the translator has been modified so that send and receive actions on these channels are translated into appropriate IF rules. The IF specifications generated by the new version of the translator are valid IF files and no changes need to be made to the model-checkers. This is an advantage as there are four model-checkers which are independently developed. Making modifications to the input language of all four tools would break the compatibility between the hlpsl2if translator and the back-ends.

The translation of send and receive actions on abstract communication channels from HLPSL into IF specifications makes use of a number of new IF fact types. These facts are used to describe the knowledge of channels, as well as to describe the ability of agents to prove who has sent and received certain messages. Although it is possible to declare these facts in the signature section of each IF file, it makes more sense to add them to the IF prelude file, which is used to describe protocol independent

functions and facts.

Unfortunately, because the model-checkers are still in development, some of them do not yet parse the prelude file, and instead assume that the standard prelude file is in use. This is unfortunate as they cannot be extended with new facts, however when this support is provided, they will be able to handle the generated IF specifications without modification. The SATMC model-checker, however, *does* parse the prelude file and requires no changes to handle the IF specifications generated by the new translator. Hopefully the three other model-checkers will provide support for the prelude file in the future, as allowing new fact types to be specified in the prelude file is an important part of the IF language.

3.3.1 Abstract Communication Channels and the HLPSL Language

As part of this research the HLPSL language has been extended to include two new concepts: the new channel types, and message parameters. These extensions allow modellers to make use of abstract communication channels. This section describes the changes made to the HLPSL language to enable modellers to use abstract communication channels in their specifications. The new HLPSL grammar can be found in appendix A.1.

Abstract communication channels are declared as constants in the same way as other "shared" information is declared. Each channel is given a name and is of type channel, but must additionally be given a channel type attribute which defines its semantics. Table 3.1 shows the appropriate channel type attribute for each abstract communication channel.

Table 3.1: Channel type attributes

Channel Type	Attribute
Dolev-Yao	dy
Over-the-air	ota
Authentic	authentic
Confidential	confidential
Non-repudiation of Origin	nro
Non-repudiation of Receipt	nrr
Non-repudiation of Receipt and Origin	nro_nrr

Figure 3.8 shows an example of how to declare an abstract communication channel:

Figure 3.8: Declaring channel constants

```
const auth_chan_AB: channel (authentic)
```

This creates an authentic channel named auth_chan_AB which can be used for sending and receiving messages. Channel constants are declared in compositional roles which are used to instantiate instances of basic roles. When this is done the channel constants are passed to each basic role as arguments. This is how roles are given shared knowledge in HLPSL. Channels are treated in the same way. A typical compositional role will instantiate several basic roles and pass them as arguments such as agent names, shared keys, common functions, and any other information which the roles are assumed to share. Figure 3.9 shows a typical composition role in HLPSL. Note that this role declares an authentic channel constant and passes this channel to the instantiation of both Alice and Bob.

Figure 3.9: Declaring channels in a compositional role

```
role Session () def=

  const
    a,b : agent,
    auth_chan_AB : channel (authentic)
    composition
        Alice(a,b,auth_chan_AB)
     /\ Bob(a,b,auth_chan_AB)
```

A basic role must specify the parameters it requires when instantiated. If the basic role requires the use of an abstract communication channel, then this will appear in the parameter list of the role. Figure 3.10 displays an example of a parameter list for a basic role. Note that it includes an authentic called auth_chan_AB.

Figure 3.10: Specifying channels as parameters of basic roles

```
role Alice (
    Auth_chan_AB : channel (authentic),
    A,B : agent
  ) def =
```

When the role Alice in figure 3.10 is instantiated by a compositional role its parameters will be given the values passed to it as arguments from the compositional role. This is how the same channel is shared between multiple roles. They are simply instantiated with the same channel constant.

In order to make use of an abstract communication channel, a basic role must use it for sending or receiving messages. This is done in a similar way to the traditional send and receive actions which are used for Dolev-Yao channels. Figure 3.11 shows an example of a receive action over a Dolev-Yao channel named Channel_1.

Figure 3.11: A receive action over a Dolev-Yao channel

```
State = 0 /\ Channel_1(Msg') =|>
...
```

Figure 3.12 shows an example of a send action over a Dolev-Yao channel named Channel_2.

Figure 3.12: A send action over a Dolev-Yao channel

```
... =|>
State' = 1 /\ Channel_2(Msg)
```

If a channel event occurs on the left hand side of a transition, then it is a receive action. If it is on the right hand side of a transition, then it is a send action.

If one of the new channel types is being used then additional information is required from the modeller. The concept of “message parameters" is introduced. Message parameters specify a to address and a from address for each message. When sending a message, the from field specifies the agent sending the message, and the to field specifies the agent which is the intended recipient of the message. When receiving a message, the from field is the expected send, and the to field is the agent receiving the message. This information is used by the abstract communication channels to reason about the source and destination of messages. For example, when a message is sent on a confidential channel, it can only be received by the agent specified in the to field of the message. Figure 3.13 shows an example of a receive action on an authentic channel named Auth_Chan_AB.

Figure 3.13: A receive action on an authentic channel

```
State = 0 /\ Auth_Chan_AB(A,B;Msg') =|>
...
```

This receive action specifies that the receiving agent believes that the message was sent from A to B. This information alone imposes no restrictions on the messages which can be received, but because this is an authentic channel, the receive event will only be enabled if an appropriate message was sent by A. A send action on an authentic channel is demonstrated in Figure 3.14. The example shows Msg being sent from A to B. Whether or not it will reach B is dependent on the channel and the intruder.

Figure 3.14: A send action on an authentic channel

```
... =|>
State' = 1 /\ Auth_Chan_AB(A,B;Msg)
```

It is important to note that in HLPSL each role is given knowledge of the other agents. This is required when inserting goal related facts such as witness and request. Modellers should be careful not to base the protocol specification on receiving this type of information but should instead create different variables with which to model agent identifiers.

All send and receive actions over the new channel types must specify message parameters as described above. The only channel type which doesn't require message parameters is the Dolev-Yao channel type. This is for backwards compatibility reasons.

3.3.2 Translating Abstract Communication Channels into IF

The precise semantics of the HLPSL language can be defined in two ways. During the design of the language its semantics were defined using TLA [29]. TLA provides a precise but theoretical definition of the HLPSL language. In practice, the semantics of the HLPSL language can also be considered in terms of the lower level term-rewriting language IF. IF is the language which HLPSL specifications are translated into for verification by the back-ends.

This sections presents the semantics of each of the new channel types in IF. It explains how send and receive actions on abstract communication channels are translated into IF term-rewriting rules by the extensions made to the hlpsl2if translator tool.

Additions to the IF Prelude File

In order to adequately express the required functionality of the new channel types in IF, it was necessary to add some new facts to the IF language. The IF language itself has not been modified, but these new facts have been added by declaring them in the signature section of the prelude file.

The new facts provide the IF language with the ability to specify who has sent what messages over which channels using the `sent` fact. The `sent` fact takes the following form: `sent: agent * message * channel -> fact`. When a message is over an abstract communication channel, a sent fact is generated which specifies the agent that sent the message, the message which was sent, and the channel on which this message was sent.

In addition to specifying who sent what, the new channel types require the ability to express who can prove what. In order to accommodate this, the `canprovesent` and `canprovereceived` facts were introduced. The `canprovesent` fact has the following form: `canprovesent: agent * agent * message -> fact`. It is generated whenever an agent gains the ability to prove that another agent sent a certain message. The first parameter of the fact specifies the agent which has the ability to prove the fact, the second parameter specifies the agent which sent the message, and the third parameter specifies the message which was sent. The `canprovereceived` fact was also added to the prelude file. It is used to express an agent's ability to prove that another agent received a message and has the following

form: `canprovereceived: agent * agent * message -> fact`. The `canprovereceived` fact is interpreted as: the agent specified in the first parameter can prove that the agent in the second parameter received the message in the third parameter.

In addition to the new facts, the `channel` type was added to the type symbols section of the prelude file. This was required because channels are now declared and used as symbols, instead of being completely replaced by IKNOWS facts. A copy of the new IF prelude file is available in appendix A.2.

The form of the proof facts `canprovesent` and `canprovereceived` is not ideal. A more general solution is described in [28]. It uses a `canprove` fact of the following form: `canprove: agent * fact -> fact`. This `canprove` fact can then be used to express an agent's ability to prove facts such as: `sent: agent * message -> fact` and `received: agent * message -> fact`. This approach is more suitable as it is more flexible. It allows agents to prove arbitrary facts instead of just `sent` and `received` facts. This solution has not been adopted at this point because the SATMC model checker will not parse IF files which contain nested facts. In the future this is expected to change and the more flexible solution will be adopted.

Dolev-Yao Channels

Dolev-Yao channels were already supported in HLPSL, but are included here for completeness. When messages are sent on a Dolev-Yao channel, they are simply redirected to the intruder. A receive action on a Dolev-Yao channel is enabled if the intruder has knowledge of an acceptable message to be received. Therefore, all send and receive actions on Dolev-Yao channels are expressed using the `iknows` fact, which describes the knowledge of the intruder. The `iknows` fact is of the form `iknows: message -> fact`. When a message is sent on a Dolev-Yao channel, the result in IF is the generation of an iknows fact. Conversely, a receive action on a Dolev-Yao channel is enabled if the intruder has knowledge of an appropriate message, and is expressed in IF by placing an `iknows` fact in the left hand side of the transition. Recall that the left hand side of an IF transition expresses facts which must be true for the transition to be enabled and the right hand side of an IF transition expresses the consequences of the transition should it be triggered.

Over-the-air Channels

Over-the-air channels deliver the message to the intruder at the same time as they deliver the message to the recipient. A send action on an over-the-air channel will send the message to the channel, but not to the intruder. A receive action on an over-the-air channel is enabled if either the channel or the intruder has knowledge of a message which the recipient will accept. Additionally, if a receive action is triggered by the channel, then the knowledge of the intruder will be updated as an after-effect.

The translation of send and receive actions on over-the-air channels into IF is more complicated than for Dolev-Yao channels. When a message is sent on an over-the-air channel, a `sent` fact is generated. For example, a send action on an over-the-air channel would generate a fact similar to: `sent(A,Msg,Channel_1)`. This means that A sent Msg on the channel Channel_1.

Receive facts on over-the-air channels are enabled if an appropriate `sent` fact exists or if there is an appropriate `iknows` fact. This "or" behaviour requires two identical IF transitions to be generated from a single HLPSL transition. One IF transition will contain an `iknows` fact on the left hand side of the transition. This transition is used to specify what happens when the intruder has knowledge of a

message which would be accepted by the recipient at this point. The other IF transition contains a `sent` fact on the left hand side, and expresses a transition which is triggered by the *channel*. Additionally, this second transition contains an `iknows` fact on the right hand side of the transition. This means that if the transition is triggered by the channel, then the knowledge of the intruder is updated with the message which was received.

Authentic Channels

Authentic channels ensure recipients that the message they are receiving was, in fact, sent by the agent the recipient expects to have sent the message. A send action on an authentic channel sends the message to the intruder *and* to the channel. A receive action on an authentic channel is only enabled if the channel has knowledge of a matching message. A send action on an authentic channel is translated by adding two facts to the right hand side of the IF transition: an `iknows` fact and a `sent` fact. In other words, the message is sent to both the intruder and to the channel. The `sent` fact is of the same form as the `sent` fact used for over-the-air channels. In fact, `sent` facts of this form are used by many of the abstract communication channels described in this chapter. A receive action on an authentic channel is translated into IF by including the `sent` fact in the left hand side of the transition. This means that a receive action on an authentic channel is only enabled if an appropriate message has been sent by the expected agent on the channel.

Confidential Channels

Confidential channels provide the sender of a message with a guarantee that the message will only be delivered to the intended recipient. A send action on a confidential channel will need the message to the channel and not to the intruder. A receive action on a confidential channel is enabled when either the intruder or the channel has knowledge of an appropriate message.

Send actions on a confidential channel do not generate an `iknows` fact. Instead they generate a `sent` fact as described previously. Receive actions on confidential channels are translated in a similar way to those of over-the-air channels. Two IF transitions are generated, one with an `iknows` fact in the left hand side of the transition, and one with a `sent` fact in the left hand side of the transition. Unlike over-the-air channels, receive actions on confidential channels do not generate `iknows` facts.

Non-repudiation of Origin Channels

Non-repudiation of origin channels generate a proof which the recipient of a message can use to prove that the sender of a message actually sent the message. All non-repudiation channels are also authentic channels. When a message is sent on a non-repudiation of origin channel it is sent to the intruder and to the channel. A receive event on a non-repudiation of origin channel is enabled if the channel possesses knowledge of a matching message. Upon receipt of a message on a non-repudiation of origin channel a proof fact is generated for the recipient.

Non-repudiation of origin channels are translated into IF in the same way as authentic channels, except that receive actions on non-repudiation of origin channels generate an additional `canprovesent` fact on the right hand side of the IF transition.

Non-repudiation of Receipt Channels

Non-repudiation of receipt channels provide the sender of a message with proof that the recipient received the message. This proof is generated upon receipt of the message by the recipient. Non-repudiation of receipt channels are identical to non-repudiation of origin channels except that they generate a `canprovereceived` fact upon receipt of messages.

Non-repudiation of Origin and Receipt Channels

Non-repudiation of origin and receipt channels provide a proof to the recipient that the sender sent the message, and a proof to the sender that the recipient received the message. They are translated as authentic channels, except that they generate two extra facts upon the receipt of messages on the channel. They generate both `canprovesent` and `canprovereceived` facts.

Chapter 4

Case Study

The previous chapter presented an extension to protocol specification languages which allows modellers to specify assumptions they would like to make about communication mediums. It described the advantages of the approach chosen, the details of the proposed solution, and how the solution was implemented.

This chapter will justify the claims made in Chapter 3. It is an experiment in the application of abstract communication channels and attempts to determine how useful these concepts are to protocol modellers. It will present three examples of how to model complex protocols using abstract communication channels. These examples will demonstrate how abstract communication channels can make life easier for protocol modellers, and how they can increase the expressive capabilities of protocol specification languages. Each of the three examples is a real protocol which is difficult to model with the tools available today.

Section one presents the first example; the Purpose Built Keys Framework [9]. Section two presents the Asokan-Shoup-Waidner Contract Signing Protocol [4], and section three presents the Internet Open Trading Protocol [11].

4.1 The Purpose Built Keys Framework

The Purpose Built Keys framework (PBK) is described in [9]. It is an unorthodox protocol because it makes an unusual assumption about the environment. The protocol assumes that the first message of the protocol is authentic. In other words, the protocol is only secure if the first message is received without being modified during transmission. The point of the PBK protocol is obviously not to provide infallible security, but rather to provide improved security in a situation where there is no existing relationship between the participants. The PBK protocol provides a security property known as "sender invariance" [7]. Sender invariance means that the source of a message is guaranteed to be consistent. If the first message is received unmodified, then sender invariance can be modelled as an authentication property.

The PBK Protocol is used when Alice and Bob have no pre-existing keys with which to authenticate each other, and when there is no Public Key Infrastructure available or suitable. The protocol establishes a public and private key for Alice, which can be used by Bob to determine whether or not messages have actually been sent by Alice. The protocol only authenticates Alice if the first message

Figure 4.1: The Purpose Built Keys Protocol

```
A -> B: A, PK_A, hash(PK_A)
A -> B: {Msg}inv(PK_A), hash(PK_A)
B -> A: Nonce
A -> B: {Nonce}inv(PK_A)
```

Figure 4.2: Expressing authentication in HLPSL

```
goal
  Bob authenticates Alice on Msg
end goal
```

is received by Bob intact. Another way of looking at this is to say that Bob cannot actually authenticate Alice, but Bob is guaranteed that the source of the messages is consistent.

The PBK framework is intended to be used by either an application level protocol, or in an on-demand manner by applications. It is described as a framework, rather than a protocol, because the specification doesn't contain enough details for independent implementations to interact. This is left up to vendors. The framework does provide, however, adequate information for a formal analysis.

The protocol is initiated by Alice when she sends an identifier, an arbitrary public key, and a hash of the public key to Bob. This is the message which must be received by Bob intact. Once this message has been received, Alice can communicate with Bob by signing messages using her public key. Figure 4.1 shows the flow of messages in the PBK framework. Note that this version of the protocol is only one way in which the specification can be concretely instantiated. There are several other ways the messages could be exchanged, but the differences are minor and unimportant.

In order to model this protocol, it is necessary to express either sender invariance, or authentication as a security goal. Fortunately, authentication is easily expressed using a HLPSL goal macro. Figure 4.2 shows how this can be done.

There are two approaches to modelling the PBK protocol. The first approach involves providing both party's with cryptographic keys and actually modelling the first message being sent in a secure manner. Alice and Bob are assumed to share a secret key. The first message is encrypted by Alice using this secret key. When Bob receives the first message, he ensures that it was encrypted with the appropriate key. This is how to model the assumption that the first message is unmodified, however this approach adds complexity to the specification as well as forcing the modeller to invent a way in which the first message can be made authentic; the secret key does not really exist!

The second approach is to explicitly model the assumptions made by the protocol using an abstract communication channel. An authentic channel is a suitable solution, as it provides the exact property that is assumed about the first message: the message sent from Alice is received unmodified by

Figure 4.3: The Environment role of the PBK specification

```
role Environment() def=

  const
    a,b : agent,
    snd,rcv : channel (dy),
    snd_s : channel (authentic),
    hash : function,
    msg_id : protocol_id,
    ip_a : text,

  knowledge(i) = {a,b,hash,ip_a}

  composition

    Session(a,b,snd,rcv,snd_s,hash,msg_id,ip_a)
  /\ Session(a,b,snd,rcv,snd_s,hash,msg_id,ip_a)
  /\ Session(i,b,snd,rcv,snd_s,hash,msg_id,ip_a)

end role
```

Bob. This approach is desirable because it removes the imaginary shared key and encryption / decryption actions from the protocol specification, and instead captures the assumption using an authentic channel.

In order to make use of an authentication channel, it must be declared as a constant in the top level role and passed to each role as an argument. Figure 4.3 shows the HLPSL code for the top level role Environment.

Each role can send or receive on this channel using the standard syntax for sending and receiving on abstract communication channels. Figure 4.4 contains the HLPSL code for the role Alice, which demonstrates the send action over the authentic channel.

Note how the first message is sent over the authentic channel SND_S. All other messages are sent and received over the Dolev-Yao channels. This means that Bob is assured that the first message received from B is authentic, while the rest of the messages can potentially be modified or even created by the intruder. This behaviour appropriately captures the assumption of the PBK protocol. The complete specification of the PBK framework is available in Appendix B.1.

In this example an Abstract Communication Channel has been used to specify an assumption made by the PBK framework: that the first message is received intact. Upon applying the SATMC model-checker to the model generated, it appears that the PBK framework does guarantee the authenticity of Msg if the first message of the protocol is received intact.

Figure 4.4: The Alice role of the PBK specification

```
role Alice (
    A,B : agent,
    SND,RCV : channel(dy),
    SND_S : channel(authentic),
    Hash : function,
    Msg_Id : protocol_id,
    IP_A : text,
    Shared_Key : symmetric_key
) played_by A def =

  local
    State : nat,
    PK_A : public_key,
    Msg : text (fresh),
    Nonce : text

  init
    State = 0

  transition

1. State  = 0 /\ RCV(start) =|>
    State' = 2 /\ SND_S({IP_A.PK_A'.Hash(PK_A')}Shared_Key)

2. State  = 2 /\ RCV(start) =|>
    State' = 4 /\ SND({Msg'}inv(PK_A).Hash(PK_A))
                  /\ witness(A,B,Msg_Id,Msg')

3. State  = 4 /\ RCV(Nonce') =|>
    State' = 6 /\ SND({Nonce'}inv(PK_A))

end role
```

Figure 4.5: The ASW exchange sub-protocol

```
O -> R: me1 = {Ko.Kr.T.Text.H(No)}inv(Ko)
R -> O: me2 = {me1.H(Nr)}inv(Kr)
O -> R: me3 = No
R -> O: me4 = Nr
```

Figure 4.6: The ASW abort sub-protocol

```
O -> T: ma1 = {ABORTED.me1}inv(Ko)
T -> O: ma2 = {me1.me2}inv(Kt)
T -> O: ma2 = {ABORTED.ma1}inv(Kt)
```

4.2 The Asokan-Shoup-Waidner Fair Exchange Protocol

The Asokan-Shoup-Waidner contract signing protocol [4] (ASW), is a fair exchange protocol which attempts to allow two parties to fairly exchange digital signatures on a pre-agreed text. The ASW protocol is an *optimistic* fair exchange protocol, which means that it makes use of a trusted third party, but only if something goes wrong with the exchange. The ASW protocol presents a challenge to modellers because of its requirements. Fairness is not a standard security property and it is only recently that it has been successfully specified [27, 46]. In addition to fairness, the ASW protocol has a non-repudiation security property: non-repudiation of origin and receipt for both the originator and the recipient. Non-repudiation security properties cannot be modelled using HLPSL, as there is no way to reason about what an agent can or cannot prove.

The ASW protocol is broken up into three sub-protocols, the exchange sub-protocol, the abort sub-protocol, and the resolve sub-protocol. If all goes according to plan and neither party backs out of the exchange, the exchange sub-protocol is the only one executed. Figure 4.5 shows the exchange protocol.

O is the originator of the protocol and R is the responder. T is the trusted third party, but is not involved in the exchange protocol. Both O and R are assumed to have knowledge of a pre-agreed text which they will sign. An ASW contract can take two forms. The first type is *me1.No.me2.Nr.* The second type of ASW contract is *{me1.me2}sig_T*.

To begin with, O sends his commitment to the exchange. This includes the hash of a nonce which he keeps secret for now. R responds with his own commitment to the exchange. He also sends the hash of a nonce and keeps the value of the nonce to himself. To complete the exchange of valid signatures, each party then sends the value of his privately generated nonce, No and Nr. Each party is now in possession of a valid contract. In this situation the protocol terminates successfully.

The originator has the power to abort the protocol. An honest originator will only do this if no response is received from the responder within a certain period of time. Figure 4.6 shows the abort

Figure 4.7: The ASW resolve sub-protocol

```
O -> T: mr1 = {me1.me2}inv(Ko)
T -> O: mr2 = {ABORTED.ma1}inv(Kt)
T -> O: mr2 = {me1.me2}inv(Kt)
OR:
R -> T: mr1 = {me1.me2}inv(Kr)
T -> R: mr2 = {ABORTED.ma1}inv(Kt)
T -> R: mr2 = {me1.me2}inv(Kt)
```

sub-protocol. The originator sends a signed abort request to the trusted third party which contains a copy of the first message of the protocol, me1. When this message is received by the trusted third party it will check that it has not previously resolved this exchange, and if it hasn't will issue an abort token to the originator. The abort token is a commitment to the originator that the trusted third party will not resolve the exchange.

Either party may attempt to resolve the protocol once they have received the other participant's commitment to the exchange. This involves sending the first and second messages of the exchange protocol to the trusted third party as evidence of the other parties commitment to the exchange. Once the trusted third party receives a resolve request it checks to see if it has already aborted this protocol run. If so, it responds with an abort token, otherwise it responds by signing me1 and me2. This message constitutes a valid contract as it contains both party's commitments to the exchange, and it is signed by the trusted third party. Figure 4.7 shows the resolve sub-protocol of the ASW protocol.

The non-repudiation property of the ASW protocol is defined in [4] as

> After an effective exchange, (i.e. P has received iQ at the end of the exchange) , P will be able to prove that iQ originated from Q, and that Q received iP

This definition is one-way. It only refers to P's security properties. A similar requirement holds for the other party.

Specifying the non-repudiation requirements of the ASW protocol can be done by using non-repudiation of origin and receipt channels and by augmenting the HLPSL specification with custom facts which can be used to reason about who has received what. The custom facts used are of the form`received: agent * message * message -> fact.`

A received fact means that the specified agent has received the first message in exchange for the second message. There is another custom fact used in the ASW specification, but this is just a workaround that is required because the hlpsl2if translator does not yet support the HLPSL accept statement. The HLPSL accept statement is supposed to be used to specify when an agent has reached a successful final state of the protocol run. It is not yet supported by the translator as it is a component of the sequential composition features of HLPSL, which are also not yet supported. The second custom fact is of the form: `accepted: agent -> fact.` This simply means that the specified agent has finished its protocol run.

Figure 4.8: The Non-Repudiation Requirements of the ASW Protocol

```
goal nro_1(O,Me1,No,Nr,R,Me2) :=
  accepted(O).
  accepted(R).
  received(O,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  received(R,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  not(canprovesent(O,R,Me2)

goal nro_2(O,Me1,No,Nr,R,Me2) :=
  accepted(O).
  accepted(R).
  received(O,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  received(R,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  not(canprovesent(O,R,Nr)

goal nrr_1(O,Me1,No,Nr,R,Me2) :=
  accepted(O).
  accepted(R).
  received(O,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  received(R,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  not(canprovereceived(O,R,Me1)

goal nrr_2(O,Me1,No,Nr,R,Me2) :=
  accepted(O).
  accepted(R).
  received(O,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  received(R,pair(Me1,pair(No,pair(Me2,Nr))),pair(Me1,pair(No,pair(Me2,Nr)))).
  not(canprovesent(O,R,No)
```

The custom facts used in the ASW specification are copied directly into the IF specification. They have no semantics in terms of HLPSL, and are simply copied from their location in a HLPSL transition to the corresponding location in the corresponding IF rule.

The non-repudiation requirements of the ASW protocol are shown in Figure 4.8:

These requirements are specified in IF, rather than HLSPL. This is because HLPSL goals are currently restricted to authentication and secrecy macros. This is a restriction of the tools. In the future the HLPSL language will allow modellers to specify goals as temporal logic style expressions, but for now, only high level secrecy and authentication macro constructs are supported. For now, goals other than secrecy and authentication must be specified at the IF level. It is necessary to translate the HLPSL specification into IF, then append a goal section to the IF file. The goal must be written in IF.

The non-repudiation requirement in Figure 4.8 specifies a set of attack states. If any of the rules is ever true, then the security requirement is violated. The first requirement can be read as follows: if

both O and R have finished, and both O and R have received a valid contract, and O cannot prove that R sent me2, then the goal nro_1 is violated. Each of the other three goals is interpreted in a similar way. The HLPSL specification of the ASW protocol makes use of non-repudiation of origin and receipt channels for communication between the originator and the recipient. This is because all the messages between those participants are digitally signed.

The full specification of the ASW protocol is available in appendix B.2. It makes use of a number of non-repudiation channels, and also uses two custom facts: received and accepted. These facts can be arbitrarily inserted into HLSPL specifications and will be copied verbatim into the generated IF file. They do not change the behaviour of the protocol, but allow IF goals to be written which make use of the custom facts

4.3 The Internet Open Trading Protocol

The Internet Open Trading Protocol (IOTP) [11] provides an interoperable framework for Internet commerce. IOTP provides Internet-based commerce models which mirror the way in which business in conducted offline. It provides frameworks for the negotiation of transactions, value exchanges, payment, delivery of goods, and receipts. All of these frameworks are designed to operate in ways familiar to the real world.

IOTP is designed to be a framework which provides global interoperability between e-commerce protocols. It is designed in a general way which makes it easy for most e-commerce protocols to conform to the IOTP standards. For instance, it encapsulates payment systems such as SET [32, 33, 34, 35], CyberCash [22], Mondex [1], DigiCash [15], and GeldKarte [2].

The IOTP protocol itself is made up of a number of exchanges which can be combined in certain ways to create transactions. Each transaction represents a traditional trading process. The transactions provided by the IOTP protocol are purchase, refund, value exchange, authentication, withdrawal, deposit, inquiry, and ping. This case study analyses an IOTP purchase transaction.

The purchase transaction consists of three exchanges: an authentication exchange, an offer exchange, and a payment exchange. Figure 4.9 shows an Alice-Bob description of the IOTP purchase transaction.

The IOTP exchanges make use of sub-protocols. The protocol is really just an encapsulating transport protocol for the e-commerce protocols which comply with it. In order to model such a protocol it is usually necessary to model specific sub-protocols as well as the main protocol. For example, the Extensible Authentication Protocol (EAP) provides a common authentication framework which can be used with a large number of authentication mechanisms. When modelling EAP one must decide which authentication mechanisms to model. Using abstract communication channels, however, the modeller can specify the EAP protocol itself, and simply make assumptions about the underlying authentication protocol. This is the approach taken to model the IOTP purchase transaction.

The purchase transaction has been modelled using an abstract communication channel. An authentic channel is used when sending the Auth_Response message in the authentication exchange. This models the assumption that the merchant and client are using an authentication protocol which authenticates the client on this message. The assumptions of the payment exchange sub-protocol are not modelled, as the payment exchange makes no security assumptions about the sub-protocol. Figure 4.10 shows the HLPSL specification for the customer role.

Figure 4.9: The IOTP Protocol

```
Authentication Exchange
  C -> M: Offer
  M -> C: Algorithm.Challenge.Trading_Role_Info_Request
  C -> M: Auth_Response.C
  M -> C: Status
Offer Exchange
  C -> M: Offer
  M -> C: {C.M.P.Order.Payment}inv(KeyM)
Payment Exchange
  C -> M: Offer
  M -> C: BrandList
  C -> M: Selection
  M -> C: {Payment.M.P}inv(KeyM)
  C -> P: {Status.{Payment.M.P}inv(KeyM)}inv(KeyC)
  C <> P: ...
  P -> C: Status.{Pay_Receipt.{Payment.M.P}inv(KeyM)}inv(KeyP)
```

If signatures are being used, IOTP requires the payment handler to verify that a payment action is actually authorised. This can be modelled as authentication of the Payment message from the customer to the payment handler. In HLPSL this is expressed as: `Payment_Handler authenticates Customer on Payment`. Upon analysis, the IOTP protocol was determined to satisfy its authentication goal. The full specification of the protocol is available in appendix B.3.

Figure 4.10: The IOTP Customer Role in HLPSL

```
role Customer (
    C,M,P                     : agent,
    Offer                     : message,
    Status                    : message,
    Trading_Role_Info_Request : message,
    KeyC,KeyM,KeyP            : public_key,
    SND,RCV                   : channel (dy),
    SND_A                     : channel (authentic)
) played_by C def =

  local
    State         : nat,
    Challenge     : text,
    Algorithm     : text,
    Order         : text,
    Payment       : text,
    BrandList     : text,
    Pay_Receipt   : text,
    Auth_Response : text (fresh),
    Selection     : text (fresh)

  init
    State = 0

  knowledge(C) = {inv(KeyC)}

  transition

 1. State  = 0  /\ RCV(start) =|>
    State' = 1  /\ SND(Offer)

 2. State  = 1  /\ RCV(Algorithm'.Challenge'.Trading_Role_Info_Request) =|>
    State' = 2  /\ SND_A(C,M;Auth_Response'.C)

 3. State  = 2  /\ RCV(Status) =|>
    State' = 3  /\ SND(Offer)

 4. State  = 3  /\ RCV({C.M.P.Order'.Payment'}__inv(KeyM)) =|>
    State' = 4  /\ SND(Offer)

 5. State  = 4  /\ RCV(BrandList') =|>
    State' = 5  /\ SND(Selection')

 6. State  = 5  /\ RCV({Payment.M.P}__inv(KeyM)) =|>
    State' = 6  /\ SND(Status.{Payment.M.P}__inv(KeyM))    %% to P
                /\ witness(C,P,payment1,Payment)

 7. State  = 6  /\ RCV(Status.Pay_Receipt'.{Payment.M.P}__inv(KeyM)) =|>
    State' = 7

end role
```

Chapter 5

Conclusions

5.1 Results

This research conducted experiments in modelling large and complicated security protocols using the state-of-the-art specification language HLPSL. The goal of these experiments was to evaluate the scope of protocols that HLPSL could express. Since HLPSL and the AVISPA tools aim to integrate protocol analysis into the design phase of protocols it is important that the language can be used to express the types of protocols currently being developed by industrial bodies.

The experiments discovered that the HLPSL language was capable of expressing most authentication and key exchange protocols, but that it was difficult to specify some other types of protocols in HLPSL. The HLPSL language only supported a single model of the intruder's capabilities. This is limiting as more and more protocols are being proposed for operating in different environments and over new types of media. It should be possible to analyse these protocols, and even just parts of these protocols, with respect to different models of an intruder's capabilities. In addition to this limitation, the HLPSL language could not be used to model non-repudiation protocols or security protocols with non-repudiation requirements. The language provides no way to specify which actions generate proof evidence, and thus there is no way to reason about an agent's ability to prove or deny its actions or the actions of other agents. Non-repudiation properties are an important part of e-commerce protocols and exchange protocols and this lack of support significantly reduces the applicability of the AVISPA tools. It was also discovered that modelling large protocols that made use of sub-protocols was difficult and time-consuming. Unfortunately there was no simple way to merge a pre-existing specification of a protocol into a larger specification as a sub-protocol. The specifications had to be manually combined into a single specification. This process was difficult and resulted in unnecessarily complex specifications. These issues were identified as key limitations of the HLPSL language which inhibited the scope of the AVISPA tools.

The next part of the research project consisted of the conceptualisation and formulation of a mechanism with which to overcome the limitations of the HLPSL language which were discovered in the experimentation phase of the project. A new modelling construct was proposed called abstract communication channels. These channels were designed to allow modellers to explicitly specify the assumptions they would like to make about messages sent and received on particular channels. The goal was to use these channels to allow modellers to overcome the limitations discovered in the experimentation phase of the project, and thus to increase the scope of the AVISPA tools.

A number of new channel types were defined which allowed modellers to specify alternative intruder models, to make assumptions about properties provided by sub-protocols, and also to express the proofs certain actions would generate. The HLPSL language was extended with declarations of these new channel types, and with message parameters that were used to specify the source and destination addresses of messages.

The semantics of each of the new channel types was defined in the IF term-rewriting language, and the tool used to translate HLSPL specifications into IF specifications was modified to capture the required behaviour of each channel. In order to accomplish this, a number of new fact types were added to the IF prelude file. These new facts were used to reason about the knowledge of a channel, the source of messages, and agent's abilities to prove things. Send and receive actions on the channels were then translated into appropriate IF rules based on these facts.

The result was a selection of channel types which allowed modellers to easily make assumptions about the properties of a communication medium. These assumptions could be related to the capabilities of an intruder, to the physical properties of a medium, or to properties provided by a sub-protocol which was being abstracted into a channel. The assumptions could also be used to specify that messages sent or received on a certain channel would generate proofs that could be used by agents as evidence that messages were sent or received.

The next stage of the project was a case study which aimed to evaluate the value of the extensions. Three protocols were chosen from among those that had presented modelling difficulties in the experimentation phase of the research. Each protocol provided an opportunity to experiment with abstract communication channels. The three protocols chosen for the case study were the Purpose Built Keys Framework (PBK), the Asokan-Shoup-Waidner Fair Exchange Protocol (ASW), and the Internet Open Trading Protocol (IOTP).

The case studies were an experiment in using abstract communication channels. On a practical level they aimed to determine if the new versions of the tools worked and whether or not the models generated were consistent with the required semantics of the new channel types. On a more fundamental level the case studies hoped to evaluate the worth of the new channel types.

The PBK framework made an assumption: that the first message of the protocol was received intact. Prior to abstract communication channels this was modelled by giving each party a shared key and encrypting the message. This approach is awkward and may not even be a correct. Using abstract communication channels the assumption was modelled by sending the first message over an authentic channel. The channel behaved as expected and the specification was simplified by removing the artifacts associated with the imaginary shared key.

The ASW protocol presented a challenge because it had non-repudiation requirements. This had previously been modelled, but the non-repudiation properties of the protocol were not checked. By sending messages which would generate proofs over non-repudiation channels it was possible to specify which actions would generate evidence. The non-repudiation properties of the ASW protocol were then specified in terms of the proof facts generated by the channels and the non-repudiation properties of the protocol were verified. This result was not possible using HLPSL prior to the addition of abstract communication channels.

The IOTP protocol provides a framework into which sub-protocols are incorporated. The purchase transaction of the IOTP protocol was specified using abstract communication channels to model the authentication sub-protocol. This allows modellers to concentrate on the IOTP protocol itself, rather than spending time choosing a particular sub-protocol and modelling it as part of the main protocol.

The IOTP purchase transaction was successfully analysed using an authentic channel to model the authentication sub-protocol.

5.2 Evaluation of the Results

At this point the evaluation of extensions is only at an initial stage. The experiments have been conducted by a single person who has experience with both the old version of the tools and the new version. More detailed evaluation results will be obtained when the new versions of the tools are merged back into the AVISPA project and a large number of modellers begin to experiment with the new channel types.

The key evaluation criteria with which to assess the overall results of the research project were identified as correctness, usefulness, applicability, ease of use, interoperability, and extensibility.

5.2.1 Correctness

Are send and receive actions correctly translated into IF rules?

For each of the new channel types, test scenarios were created which would verify whether or not the channel was acting in the appropriate manner. For example, a protocol was modelled where an agent sent a message in clear text over a confidential channel. If the secrecy of the message was violated, or the IF file generated was invalid, then the channel was clearly not operating properly. The SATMC back-end was used for this validation process. In addition, the IF files generated were manually examined to ensure the translation was correct. In all cases, the bugs found were corrected. The successful analysis of the three case studies suggests that the final implementation is correct.

5.2.2 Usefulness

Did the new channel types overcome the limitations identified during the experimentation phase of the project?

In each of the case studies the use of abstract communication channels improved the specification process. For the PBK protocol, abstract communication channels allowed the assumption on the first message to be explicitly modelled, instead of by adding artificial additions to the specification, abstract communication channels allowed the ASW protocol's non-repudiation requirements to be expressed in terms of the proof facts generated by the channels, and the specification of IOTP protocol was simplified by modelling the authentication sub-protocol as an authentic channel. It is clear that abstract communication channels improved the specification process for the protocols in the case study and were therefore useful for modelling those protocols.

5.2.3 Applicability

Are the new channel types useful for a wide range of protocols? How much was the scope of the AVISPA tools increased?

The protocols in the case study were hand-picked because they provided opportunities to experiment with abstract communication channels. The applicability of abstract communication channels questions whether or not they will be useful for *other* protocols. It is expected that abstract communication channels will be useful for a large proportion of protocols which might be modelled, particularly real-world protocols which are often large and make use of sub-protocols, however further untargeted experimentation is required before this theory can be verified.

5.2.4 Ease-of-use

Were the changes to the HLPSL language minimised? Do they complicate the specification process?

The changes to the HLPSL language are minimal. They require the modeller to declare the channel in the same way as before, but with a different type attribute. In addition, message parameters must be specified when sending and receiving messages over the new channel types. This is a simple process and does not significantly complicate the HLPSL language.

The specification process itself has been simplified. If a protocol does not require the use of abstract communication channels, it can be modelled in the same way as before. If the protocol makes an assumption, or if it makes use of sub-protocols, then the specification process can be simplified by using abstract communication channels. The time saving is dependant on the protocol and its sub-protocols. If a protocol makes use of a single sub-protocol, then the time saving is equivalent to the amount of time it would take to model the sub-protocol. The sub-protocols in question are usually not very large protocols. The time saving per sub-protocol can roughly be estimated as half a working day for small and simple sub-protocols, and a number of working days for more complicated sub-protocols.

5.2.5 Interoperability

Did the changes made to the HLPSL language invalidate old specifications? Do the back-ends need to be modified?

Old HLPSL specifications are still valid in the new version of the language. Dolev-Yao channels are declared in the same manner as before, and send and receive actions over Dolev-Yao channels do not require any message parameters. This means that the changes to the HLPSL language only effect modellers who would like to make use of abstract communication channels.

The channels were implemented in the translation from HLPSL to IF. The IF files generated are valid IF specifications, and the model checkers should not need to be modified. This holds for the SATMC model checker, however the other model checkers (OFMC, CL-Atse, and TA4SP) do not yet fully support the IF language, and will need to provide more complete support before they will handle the IF specifications generated when using abstract communication channels. This is a limitation of the back-ends; full support for the IF language should eventually be built into the model-checkers anyway.

5.2.6 Extensibility

Were the channels added in a way that would allow more channel types to be easily added?

In terms of the HLPSL language itself, the channels have been implemented in an extensible manner. New channel types can be added to the language via the channel type attribute when declaring them. However it is necessary to manually implement the required behaviour of new channel types by modifying the hlpsl2if translator. A more extensible approach would be for modellers to define the semantics of each new channel types in the IF prelude file. It is unclear if this is feasible at this point. A format for specifying the consequences of send and receive actions would need to be devised, and the hlpsl2if translator modified to parse the IF prelude file. The approach taken is extensible, but not easily extensible.

5.3 Future Directions

5.3.1 Other types of channels

There are a number of channel types which would be useful to modellers but which were not added to the AVISPA tools because the IF language does not have support for fairness constraints. Fairness constraints allow analysts to specify that something must *eventually* happen. Some specification languages do support fairness constraints and could make use of channels such as resilient channels, which guarantee to eventually deliver any messages sent on them to the appropriate recipient.

There are some other channel types which could be useful to modellers. In fact, there are innumerable properties and combinations of properties which could be supplied to the modeller. For example confidential Authentic channels, strong authentic channels, confidential NRO channels, and authentic over-the-air channels.

It is impossible to provide a priori support for every possible type of communication channel, but as new channel types are required, they can be added to the tools. This is expected to occur as protocols diversify and are applied to different application areas. Unfortunately this process requires a developer to manually implement the channels in the hlpsl2if translator. A useful topic of future research may be to investigate a way of adding arbitrary channel types using the IF prelude file to define their semantics.

5.3.2 Composability of Protocol Specifications

The composition of simple protocol specifications into larger protocols is a current topic of interest to researchers. The principle is similar to object oriented programming. Different parts of a specification should be written separately so that they can be re-used. Furthermore, once a protocol is modelled other protocols may make use of it as a sub-protocol. The composition of *roles*, which HLPSL currently supports, allows modellers to specify that roles can be composed sequentially, or in parallel. In most cases, this does not allow specified protocols to be easily inserted into other protocol specifications. This is because when a protocol is used as a sub-protocol, it is not usually done in a sequential way, and certainly not in a parallel way. The relationship between a protocol and its assumed sub-protocol or sub-protocols is more complex. It typically involves some messages of one protocol being used in another, or values established and exchanged in the sub-protocol being used in the main protocol. It is difficult to define the relationship between a protocol specification and the specification of its sub-protocol in terms of HLPSL, however, it is possible to examine the security

properties provided by a sub-protocol, and then model the sub-protocol as an abstract communication channel.

When a specification is written for a security protocol, the analyst includes the security requirements of the protocol. A model checker of some description is then used to check that these requirements are met. If they are, then the protocol can be abstracted as an exchange of messages and a security property or security properties which apply to these messages.

A larger protocol that makes use of this sub-protocol can include the full sub-protocol as additional exchanges of messages. However in some cases it is possible to abstract a whole protocol into a simple channel that provides a certain property. An apt example of this is a protocol that authenticates Alice to Bob on a certain message or messages. A larger protocol does not need to model this protocol as a sub-protocol; it can merely use an authentic channel between Alice and Bob for sending the appropriate messages.

In some situations it may be more difficult to establish the appropriate channel type to represent a sub-protocol, especially if the various channel types are restricted in some way. However the types of available channels do not need to be limited. There are an endless variety of properties that a security protocol might provide and in order to generate channel types based on the requirements of sub-protocols an extensible channel type mechanism is required. For now this does not exist and new channel types must be programmed into the tools manually.

The biggest difficulty with this approach is automatically generating a channel type from the specification. In fact, it may not be possible to do so when the requirements are specified as temporal formulae. However when macro constructs with clear meanings are used, the process is indeed viable. An example is the standard authentication goal used in the HLPSL language. E.g,

```
Bob authenticates Alice on Msg
```

It is also important to note that the modeller may not wish to specify the sub-protocol, or even to choose a sub-protocol. Some protocols merely specify the required properties of a sub-protocol, and leave the actual choice of protocol to the implementer. A protocol developer may wish to experiment with a protocol without having to model the specifics of sub-protocols, merely their properties. In all these cases abstract communication channels can be used to model the properties of the sub-protocol and any necessary messages exchanged by this sub-protocol.

Bibliography

[1] Home page, 1996. http://www.mondex.com.

[2] Home page, 1997. http://www.gdm.de.

[3] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The aviss security protocol analysis tool. In *Computer-Aided Verification CAV'02*, Lecture Notes in Computer Science 2404, pages 349–353. Springer-Verlag, 2002.

[4] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998.

[5] AVISPA. Deliverable 2.1: The High-Level Protocol Specification Language. Available at `http://www.avispa-project.org`, 2003.

[6] AVISPA. Deliverable 2.3: The Intermediate Format. Available at `http://www.avispa-project.org`, 2003.

[7] AVISPA. Deliverable 6.1: List of selected problems. Available at `http://www.avispa-project.org`, 2003.

[8] Balfanz, Dirk, D. K. Smetters, P. Stewart, and H. C. Wong. Trusting strangers: Authentication in ad-hoc wireless networks. *Network and Distributed Systems Security Symposium*, 2002.

[9] S. Bradner, A. Mankin, and J. I. Schiller. A framework for purpose built keys (PBK). Internet draft, November 2002.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[11] D. Burdette and M. Goncalves. *Internet Open Trading Protocol*. McGraw-Hill Professional, 2000.

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

[13] R. Chadha. *A Formal Analysis of Exchange of Digital Signatures*. PhD thesis, University of Pennsylvania, August 2003.

[14] R. Chadha, M. Kanovich, and A. Scedrov. Inductive methods and contract-signing protocols. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 176–185, 2001.

[15] D. Chaum. Achieving electronic privacy. In *Scientific American*, pages 96–100, august 1992.

[16] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drieslma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Automated Software Engineering. Proceedings of the Workshop on Specification and Automated Processing of Security Requirements, SAPS'04*, pages 193–205. Austrian Computer Society, Austria, September 2004.

[17] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drieslma, J. Mantovani, S. Mödersheim, and L. Vigneron. *A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols*, volume 180 of *Automated Software Engineering*, pages 193–205. Austrian Computer Society, Austria, September 2004.

[18] J. Clark and J. Jacob. A survey of authentication protocol literature : Version 1.0., November 1997.

[19] S. Creese, M. Goldsmith, B. Roscoe, and I. Zakiuddin. Authentication for pervasive computing. In *International Conference on Security in Pervasive Computing*, pages 116–129, 2003.

[20] J. Cuellar, I. Wildgruber, and D. Barnard. The temporal logic of transitions. In *Formal Methods Europe*, 1994.

[21] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983. Also STAN-CS-81-854, May 1981, Stanford U.

[22] D. Eastlake, B. Boesch, S. Crocker, and M. Yesil. RFC 1898: CyberCash Credit Card Protocol Version 0.8. 1996.

[23] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why a security protocol is correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171. IEEE Computer Society Press, New York, May 1998.

[24] F. Gärtner, H. Pagnia, and H. Vogt. Approaching a formal definition of fairness in electronic commerce. In *Proceedings of the International Workshop on Electronic Commerce*, Lausanne, Switzerland, October 1999.

[25] P. Hankes Drielsma and S. Mödersheim. The asw protocol revisited: A unified view. In *Proceedings of the IJCAR04 Workshop ARSPA*, 2004. To appear in ENTCS, available at `http://www.avispa-project.org`.

[26] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.

[27] S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. *Journal of Computer Security*, 11(3):399–430, 2003.

[28] P. Krishnan and R. Bannerman. Verifying non-repudiation and similar properties. In *International MultiConference in Computer Science 2003 (SAM Conference)*, 2003.

[29] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[30] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56(3):131–136, november 1995.

[31] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998. See `http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/`.

[32] Mastercard and VISA. SET Secure Electronic Transaction: External Interface Guide, May 1997.

[33] Mastercard and VISA. SET Secure Electronic Transaction Specification: Business Description, May 1997.

[34] Mastercard and VISA. SET Secure Electronic Transaction Specification: Formal Protocol Definition, May 1997.

[35] Mastercard and VISA. SET Secure Electronic Transaction Specification: Programmer's Guide, May 1997.

[36] K. L. McMillan. *Symbolic model checking*. Kluwer, Dordrecht, 1993.

[37] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of the DARPA Information and Survivability Conference and Exposition: DISCEX 2000*, pages 237–250. IEEE Computer Society Press, January 2000.

[38] J. Millen and G. Denker. CAPSL and MuCAPSL. *Journal of Telecommunications and Information Technology*, (4):16–27, 2002.

[39] J. Millen and G. Denker. MuCAPSL. In *DISCEX III, DARPA Information Survivability Conference and Exposition*, pages 238–249. IEEE Computer Society, 2003.

[40] J. K. Millen. Capsl: Common authentication protocol specification language. In *Proceedings of the 1996 workshop on New security paradigms*, page 132, 1996.

[41] D. Nessett. A critique of the Burrows, Abadi and Needham logic. *ACM Operating Systems Review*, 24(2):35–38, April 1990.

[42] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[43] L. C. Paulson. *Isabelle: a Generic Theorem Prover*. LNCS 828. Springer-Verlag, 1994.

[44] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society Press, 1997.

[45] V. Shmatikov and J. Mitchell. Finite-state analysis of two contract signing protocols. *Special issue of Theoretical Computer Science on security*, 2001. Accepted for publication.

[46] V. Shmatikov and J. C. Mitchell. Analysis of a fair exchange protocol. In *Proceedings of the 1999 FLoC Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[47] P. C. van Oorschot. An alternate explanation of two ban-logic "failures". In *EUROCRYPT '93: Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 443–447. Springer-Verlag New York, Inc., 1994.

Appendix A

Additions to the AVISPA Tool

This appendix contains the new HLPSL grammar with support for declaring channel types and using parameters in send and receive actions. It also contains the modified IF prelude file, which contains the fact signatures used by the new channel types.

A.1 The new HLPSL Grammar

```
%-------------------------------------------------
%  Grammar of the HLPSL
%-------------------------------------------------
% General conventions:
% * variables (var_ident) start with a capital letter
% * constants (const_ident) are integers or start with lowercase
% * comments (%...) will be intercepted by the lexical analysis
% * Grammatical elements whose names start with "Maybe_" are optional
SpecHLPSL ::=
  Role_definitions
  Maybe_goal_declaration
  % Call of the main role (ex: Environment() )
  var_ident "(" ")"

Role_definitions ::=
  Role_definition
| Role_definition Role_definitions

% Roles may be either basic or compositional.
Role_definition ::=
  Basic_role
| Composition_role

% Basic roles must include a player definition and generally
% contain a transition declaration section.
Basic_role ::=
```

```
  "role"
  Role_header Player Role_declarations
  Maybe_transition_declaration
  "end role"

% Composition roles have no transition section, but rather
% a composition section in which they instantiate other roles.
Composition_role ::=
  "role"
  Role_header Role_declarations
  Maybe_composition_declaration
  "end role"

% A role header consists of the keywork "role", the role name
% and a list of formal argument, if any
Role_header ::=
  var_ident "(" Maybe_formal_arguments ")"

% A role's declarations include:
% * the "def=" keyword
% * (optionally) declaration of local variables
% * (optionally) declaration of any owned variables
% * (optionally) declaration of constants
% * (optionally) initalisation of variables
% * (optionally) definition of accepting states
% * (optionally) knowledge declaration(s)
Role_declarations ::=
  "def="
  Maybe_local_declaration
  Maybe_owns_declaration
  Maybe_const_declaration
  Maybe_init_declaration
  Maybe_accept_declaration
  Maybe_knowledge_declaration

% Definition of the arguments of a role.
Maybe_formal_arguments ::=
  Formal_arguments
|

Formal_arguments ::=
| Variable_declaration
| Variable_declaration "," Formal_arguments

% Used to bind the role and the identifier of the agent playing the role.
Player ::=
  "played_by" var_ident
```

```
% Declaration of local variables.
Maybe_local_declaration ::=
  "local" Variables_declaration_list
|

% owns -- That a variable is "owned" by a role means that the value of
% this variable can only be changed as specified by this role.
% Subroles instantiated by the owning role may change the variable but
% may not own it themselves.
Maybe_owns_declaration ::=
  "owns" Variables_list
|

% Declaration of constants.
Maybe_const_declaration ::=
  "const" Constants_declaration_list
|

Maybe_init_declaration ::=
  "init" Init_declarations
|

Init_declarations ::=
  Init_declaration
| Init_declaration "/\" Init_declarations

Init_declaration ::=
  Stutter_expression "=" Stutter_expression
  % Initialisation of all the elements of a set/ list/ partial function
| "/\" "_" "{" Parameters_instance "}" Init_declaration

% Acceptance is used for sequential composition to mark the stop states
% after which the following instantiation may begin.
Maybe_accept_declaration ::=
  "accept" Predicates
|

% Declaration of required knowledge
Maybe_knowledge_declaration ::=
  Knowledge_declaration Maybe_knowledge_declaration
|

Knowledge_declaration ::=
  "knowledge" "(" Variable_or_constant ")" "=" "{" Stutter_expressions_list "}"

% Definition of the transition section (for basic roles)
```

```
Maybe_transition_declaration ::=
  "transition" Transitions_list
| "transition"

% Definition of the composition section (for composed roles)
Maybe_composition_declaration ::=
  "composition" Compositions_list
| "composition"

Maybe_goal_declaration ::=
  Goal_declaration
|

% Goal_declaration defines the goals section
Goal_declaration ::=
  "goal" Goal_formulas_list "end goal"

Goal_formulas_list ::=
  Goal_formula
| Goal_formula Goal_formulas_list

Goal_formula ::=
  % Secrecy goals
  "secrecy_of" Stutter_expressions_list

  % var_ident here are only basic roles identifiers
  % "authenticates" is a shortcut for the LTL formula:
  % request (A,B,C,D) -> <-> witness(A,B,C,D)
| var_ident "authenticates" var_ident "on" Stutter_expressions_list

  % Weak authentication
| var_ident "weakly" "authenticates" var_ident "on" Stutter_expressions_list

  % General LTL formulae
| LTL_formula

Compositions_list ::=
  Composition
| Composition "/\" Bracketed_par_compositions_list
| Composition ";" Bracketed_seq_compositions_list
| "(" Compositions_list ")"

Composition ::=
| "/\" "_" "{" Parameters_instance "}" Bracketed_compositions_list
| Role_instance

Bracketed_par_compositions_list ::=
```

```
  Composition
| Composition "/\" Bracketed_par_compositions_list
| "(" Compositions_list ")"

Bracketed_seq_compositions_list ::=
  Composition
| Composition ";" Bracketed_seq_compositions_list
| "(" Compositions_list ")"

Bracketed_compositions_list ::=
  Composition
| "(" Compositions_list ")"

Role_instance ::=
  Role_instantiation
  % Loops: not translated yet.
| "LOOP" Role_instantiation
| "LOOP" "(" Role_instantiation ")"

Role_instantiation ::=
  var_ident "(" Maybe_effective_arguments ")"

Variables_declaration_list ::=
  Variable_declaration
| Variable_declaration "," Variables_declaration_list

Variable_declaration ::=
  Variables_list ":" Type_of Maybe_type_attribute

Constants_declaration_list ::=
  Constant_declaration
| Constant_declaration "," Constants_declaration_list

Constant_declaration ::=
  Constants_list ":" Simple_type_of Maybe_type_attribute

% attributes qualifying certain types of variables
Maybe_type_attribute:
  % for declaring a variable that will have a newly generated value
  "(" "fresh" ")"
  % Dolev-Yao channels
| (" "dy" ")"
  % Over-the-Air channels
| (" "ota" ")"
  % Authentic channels
| (" "authentic" ")"
  % Confidential channels
```

```
| (" "confidential" ")"
  % Non-repudiation of Origin channels
| (" "nro" ")"
  % Non-repudiation of Receipt channels
| (" "nrr" ")"
  % Non-repudiation of Origin and Receipt channels
| (" "nro_nrr" ")"
|

Type_of ::=
  Subtype_of
| Subtype_of "->" Type_of

Subtype_of ::=
  Simple_type
| "(" Subtype_of ")"
| Compound_type

Compound_type ::=
  Subtype_of "." Subtype_of
| Subtype_of "list"
| Subtype_of "set"
| "{" Subtype_of "}" "_" Bracketed_subtype_of
%| "function" "(" Subtype_of ")" % not implemented yet
| "inv" "(" Subtype_of ")"

Bracketed_subtype_of ::=
  Simple_type
| "(" Subtype_of ")"

Simple_type_of ::=
  Simple_subtype_of
| Simple_subtype_of "->" Simple_type_of

Simple_subtype_of ::=
  Simple_type
| "(" Simple_types_list ")"

Simple_types_list ::=
  Simple_subtype_of
| Simple_subtype_of "." Simple_types_list

Simple_type ::=
  "agent"
| "channel"
| "public_key"
| "symmetric_key"
```

```
| "text"
| "message"
| "protocol_id"
| "nat"
| "bool"
% hash is synonymous for function
| "function" | "hash"
| "{" Constants_or_nat_list "}"

Constants_or_nat_list ::=
  const_ident
| nat_ident
| const_ident "," Constants_or_nat_list
| nat_ident "," Constants_or_nat_list

Transitions_list ::=
  Transition
| Transition Transitions_list

Transition ::=
  % Spontaneous actions are enabled when the state predicates on the
  % LHS are true.
  Label "." Predicates "--|>" Actions

  % Immediate reactions fire immediately whenever the non stutter
  % events on the LHS are true.
| Label "." Events "=|>" Reactions

Predicates ::=
  Predicate
| Predicate "/\" Predicates

Predicate ::=
  Stutter_formula
| Variable_or_constant "(" ")"
| Variable_or_constant "(" Stutter_expressions_list ")"
| Variable_or_constant "(" Variable_or_constant "," Variable_or_constant ";" Stutter_exp
  % Dummy start message for Dolev-Yao models.
| "start" "(" ")"

Events ::=
  Predicate
| Event
| Predicate "/\" Events
| Event "/\" Events

Event ::=
```

```
  Non_stutter_formula
| Variable_or_constant "(" Non_stutter_expressions_list ")"
  % Dummy start message for Dolev-Yao models.
| var_ident "(" "start" ")"

Stutter_formula ::=
  Stutter_expression "=" Stutter_expression
| Stutter_expression "<=" Stutter_expression
| "in" "(" Stutter_expression "," Stutter_expression ")"
| "in" "(" Non_stutter_expression "," Stutter_expression ")"
| "not" "(" Stutter_expression "=" Stutter_expression ")"
| "not" "(" Stutter_expression "<=" Stutter_expression ")"
| "not" "(" "in" "(" Stutter_expression "," Stutter_expression ")" ")"
| "not" "(" "in" "(" Non_stutter_expression "," Stutter_expression ")" ")"
  % Syntactic sugar for inequality
| Stutter_expression "/=" Stutter_expression
| "(" Stutter_formula ")"

Non_stutter_formula ::=
  Non_stutter_expression "=" Stutter_expression
| Stutter_expression "=" Non_stutter_expression
| Non_stutter_expression "=" Non_stutter_expression
| Non_stutter_expression "<=" Stutter_expression
| Stutter_expression "<=" Non_stutter_expression
| Non_stutter_expression "<=" Non_stutter_expression
| "in" "(" Non_stutter_expression "," Non_stutter_expression ")"
| "in" "(" Stutter_expression "," Non_stutter_expression ")"
| "not" "(" Non_stutter_expression "=" Stutter_expression ")"
| "not" "(" Stutter_expression "=" Non_stutter_expression ")"
| "not" "(" Non_stutter_expression "=" Non_stutter_expression ")"
| "not" "(" Non_stutter_expression "<=" Stutter_expression ")"
| "not" "(" Stutter_expression "<=" Non_stutter_expression ")"
| "not" "(" Non_stutter_expression "<=" Non_stutter_expression ")"
| "not" "(" "in" "(" Non_stutter_expression "," Non_stutter_expression ")" ")"
| "not" "(" "in" "(" Stutter_expression "," Non_stutter_expression ")" ")"
  % Syntactic sugar for inequality
| Non_stutter_expression "/=" Stutter_expression
| Stutter_expression "/=" Non_stutter_expression
| Non_stutter_expression "/=" Non_stutter_expression
| "(" Non_stutter_formula ")"

Stutter_expressions_list ::=
  Stutter_expression
| Stutter_expression "," Stutter_expressions_list

Non_stutter_expressions_list ::=
  Non_stutter_expression
```

```
| Non_stutter_expression "," Expressions_list
| Stutter_expression "," Non_stutter_expressions_list

Expressions_list ::=
  Expression
| Expression "," Expressions_list

Actions ::=
  Action
| Action "/\" Actions

Reactions ::=
  Actions

Action ::=
  var_ident "'" "=" Expression
% For updating flexible functions: (not implemented yet)
%| var_ident "'" "(" Expressions_list ")" "=" Expression
| Variable_or_constant "(" ")"
| Variable_or_constant "(" Expressions_list ")"
| Variable_or_constant "(" Variable_or_constant "," Variable_or_constant ";" Expressions

Parameters_instance ::=
  Concatenated_variables_list
| "in" "(" Concatenated_variables_list "," Stutter_expression ")"

Concatenated_variables_list ::=
  Concatenated_variables
| "(" Concatenated_variables ")"

Concatenated_Variables ::=
  var_ident Var_param
| var_ident Var_param "." Concatenated_Variables

Variables_list ::=
  var_ident Var_param
| var_ident Var_param "," Variables_list

Constants_list ::=
  const_ident
| const_ident "," Constants_list

Var_param ::=
  "(" var_ident ")"
|

% We allow composed messages as effective arguments
```

```
Maybe_effective_arguments ::=
  Expressions_list
|

Variable_or_constant ::=
  var_ident
| const_ident

Variable_or_constant_or_nat ::=
  var_ident
| const_ident
| nat_ident

Stutter_expression ::=
  "(" Stutter_expression ")"
| Variable_or_constant_or_nat
| "inv" "(" Stutter_expression ")"
  % Concatenation, right-associative
| Stutter_expression "." Stutter_expression
  % List
| "[" "]"
| "[" Stutter_expressions_list "]"
  % Function application
| Variable_or_constant "(" Stutter_expressions_list ")"
  % Set
| "{" "}"
| "{" Stutter_expressions_list "}"
  % Encryption
| "{" Stutter_expression "}" "_" Bracketed_stutter_expression

Non_stutter_expression ::=
  "(" Non_stutter_expression ")"
| var_ident "'"
| "inv" "(" Non_stutter_expression ")"
  % Concatenation, right-associative
| Non_stutter_expression "." Stutter_expression
| Stutter_expression "." Non_stutter_expression
| Non_stutter_expression "." Non_stutter_expression
  % List
| "[" Non_stutter_expressions_list "]"
  % Function application
| Variable_or_constant "(" Non_stutter_expression_list ")"
  % Operator for insertion into a set or concatenation at the head of a list
| "cons" "(" Expression "," Expression ")"
  % Set
| "{" Non_stutter_expressions_list "}"
  % Encryption
```

```
| "{" Non_stutter_expression "}" "_" Bracketed_expression
| "{" Stutter_expression "}" "_" Bracketed_non_stutter_expression

Expression ::=
  Stutter_expression
| Non_stutter_expression

Bracketed_stutter_expression ::=
  "inv" "(" Stutter_expression ")"
| Variable_or_constant "(" Stutter_expressions_list ")"
| Variable_or_constant_or_nat
| "(" Stutter_expression ")"

Bracketed_non_stutter_expression ::=
  var_ident "'"
| "inv" "(" Non_stutter_expression ")"
| Variable_or_constant "(" Non_stutter_expressions_list ")"
| "(" Non_stutter_expression ")"

Bracketed_expression ::=
  Bracketed_stutter_expression
| Bracketed_non_stutter_expression

Label ::=
  const_ident
| nat_ident

% The syntax for general LTL goals is not yet fully agreed upon
LTL_formula ::=
  string

%end of grammar

%% Lexical entities:
var_ident: [A-Z][A-Za-z0-9_]*
const_ident: [a-z][A-Za-z0-9_]*
nat_ident: [0-9]+
string: \"[a-zA-Z0-9_=><-,.{}()\[\]\n\t ]+\"

%% Ignored:
comments: %[^\n]*
spaces: [\n\t ]

%end
```

A.2 The New IF Prelude File

```
section typeSymbols:

  agent, text, symmetric_key, public_key, function, table,
  message, fact, nat, protocol_id, channel

section signature:

   message > agent
   message > nonce
   message > symmetric_key
   message > public_key
   message > function
   message > table
   message > set

  pair     : message * message -> message
  crypt    : message * message -> message
  inv      : message -> message
  scrypt   : message * message -> message
  exp      : message * message -> message
  xor      : message * message -> message
  apply    : message * message -> message
  func     : public_key * nonce -> nonce

  iknows   : message -> fact
  contains : message * message -> fact
  witness  : agent * agent * protocol_id * message  -> fact
  request  : agent * agent * protocol_id * message * nat -> fact
  wrequest : agent * agent * protocol_id * message * nat -> fact
  secret   : message * agent -> fact

  sent     : agent * message * channel -> fact
  canprovesent : agent * agent * message -> fact
  canprovereceived : agent * agent * message -> fact

section types:

  PreludeK,PreludeM,PreludeM1,PreludeM2,PreludeM3 : message

section equations:

  pair(PreludeM1,pair(PreludeM2,PreludeM3)) =
    pair(pair(PreludeM1,PreludeM2),PreludeM3)

  inv(inv(PreludeM)) = PreludeM
```

```
  exp(exp(PreludeM1,PreludeM2),PreludeM3) = exp(exp(PreludeM1,PreludeM2),PreludeM3)
  exp(exp(PreludeM1,PreludeM2),inv(PreludeM2)) = PreludeM1

  xor(PreludeM1,xor(PreludeM2,PreludeM3)) = xor(xor(PreludeM1,PreludeM2),PreludeM3)
  xor(PreludeM1,PreludeM2) = xor(PreludeM2,PreludeM1)
  xor(xor(PreludeM1,PreludeM1),PreludeM2) = PreludeM2


section intruder:

  % generate rules

  step gen_pair (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(pair(PreludeM1,PreludeM2))
  step gen_crypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(crypt(PreludeM1,PreludeM2))
  step gen_scrypt (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(scrypt(PreludeM1,PreludeM2))
  step gen_exp (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(exp(PreludeM1,PreludeM2))
  step gen_xor (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(xor(PreludeM1,PreludeM2))
  step gen_apply (PreludeM1,PreludeM2) :=
    iknows(PreludeM1).iknows(PreludeM2) => iknows(apply(PreludeM1,PreludeM2))

  % analysis rules

  step ana_pair (PreludeM1,PreludeM2) :=
    iknows(pair(PreludeM1,PreludeM2)) => iknows(PreludeM1).iknows(PreludeM2)
  step ana_crypt (PreludeK,PreludeM) :=
    iknows(crypt(PreludeK,PreludeM)).iknows(inv(PreludeK)) => iknows(PreludeM)
  step ana_scrypt (PreludeK,PreludeM) :=
    iknows(scrypt(PreludeK,PreludeM)).iknows(PreludeK) => iknows(PreludeM)

  % Generating new constants of any type:

  step generate (PreludeM) :=
    =[exists PreludeM]=> iknows(PreludeM)
```

Appendix B

Protocol Specifications

B.1 The Purpose Built Keys Protocol (PBK)

```
%------------------------------------------------------------------%
%  Purpose Built Key Framework
%------------------------------------------------------------------%
%
%  A -> B: A, PK_A, hash(PK_A)
%  A -> B: {Msg}inv(PK_A), hash(PK_A)
%  B -> A: Nonce
%  A -> B: {Nonce}inv(PK_A)
%
%------------------------------------------------------------------%
%
% - The first message is assumed to be un-modified.
% - B authenticates A on Msg
%
%------------------------------------------------------------------%

role Alice (
    A,B         : agent,
    SND,RCV     : channel(dy),
    SND_S       : channel(authentic),
    Hash        : function,
    Msg_Id      : protocol_id,
    IP_A        : text
) played_by A def =

  local
    State       : nat,
    PK_A        : public_key,
    Msg         : text (fresh),
    Nonce       : text
```

```
  init
    State = 0

  transition

 1. State  = 0 /\ RCV(start) =|>
    State' = 2 /\ SND_S(A,B;IP_A.PK_A'.Hash(PK_A'))

 2. State  = 2 /\ RCV(start) =|>
    State' = 4 /\ SND({Msg'}_inv(PK_A).Hash(PK_A))
               /\ witness(A,B,Msg_Id,Msg')

 3. State  = 4 /\ RCV(Nonce') =|>
    State' = 6 /\ SND({Nonce'}_inv(PK_A))

end role

%-------------------------------------------------------------------%

role Bob (
    B,A        : agent,
    SND,RCV    : channel(dy),
    RCV_S      : channel (authentic),
    Hash       : function,
    Msg_Id     : protocol_id
) played_by B def =

  local
    State      : nat,
    Nonce      : text (fresh),
    Msg        : text,
    PK_A       : public_key,
    IP_A       : text

  init
    State = 1

  transition

 1. State  = 1 /\ RCV_S(A,B;IP_A'.PK_A'.Hash(PK_A')) =|>
    State' = 3

 2. State  = 3 /\ RCV({Msg'}_inv(PK_A).Hash(PK_A)) =|>
    State' = 5 /\ SND(Nonce')

 3. State  = 5 /\ RCV({Nonce}_inv(PK_A)) =|>
```

```
    State' = 7
               /\ request(B,A,Msg_Id,Msg)

end role

%-------------------------------------------------------------%

role Session(
    A,B         : agent,
    SND,RCV     : channel (dy),
    SND_S,RCV_S : channel (authentic),
    Hash        : function,
    Msg_Id      : protocol_id,
    IP_A        : text
) def=

  composition
    Alice(A,B,SND,RCV,SND_S,Hash,Msg_Id,IP_A)
 /\ Bob(B,A,SND,RCV,RCV_S,Hash,Msg_Id)

end role
%-------------------------------------------------------------%

role Environment() def=

  const
    a,b        : agent,
    snd,rcv    : channel (dy),
    snd_s      : channel (authentic),
    rcv_s      : channel (authentic),
    hash       : function,
    msg_id     : protocol_id,
    ip_a       : text

  knowledge(i) = {a,b,hash,ip_a}

  composition
    Session(a,b,snd,rcv,snd_s,rcv_s,hash,msg_id,ip_a)
 /\ Session(a,b,snd,rcv,snd_s,rcv_s,hash,msg_id,ip_a)
 /\ Session(i,b,snd,rcv,snd_s,rcv_s,hash,msg_id,ip_a)
 /\ Session(a,i,snd,rcv,snd_s,rcv_s,hash,msg_id,ip_a)

end role

%-------------------------------------------------------------%

goal
```

```
  Bob authenticates Alice on Msg

end goal

%-----------------------------------------------------------------%

Environment()

%-----------------------------------------------------------------%
```

B.2 The Asokan-Shoup-Waidner Contract Signing Protocol (ASW)

```
%------------------------------------------------------------%
% ASW as presented by Shmatikov and Mitchell        11/6/04
%------------------------------------------------------------%
% O and R are assumed to know text,T,Ko,Kr,Kt,ABORTED
% T knows ABORTED,T
% i knows O,R,T,Ko,Kr,Kt,H,ABORTED
%------------------------------------------------------------%
% O -> R: me1 = {Ko.Kr.T.Text.H(No)}inv(Ko)  % exchange:
% R -> O: me2 = {me1.H(Nr)}inv(Kr)
% O -> R: me3 = No
% R -> O: me4 = Nr
%------------------------------------------------------------%
% O -> T: ma1 = {ABORTED.me1}inv(Ko)         % abort:
% T -> O: ma2 = {me1.me2}inv(Kt)             % resolved?
% T -> O: ma2 = {ABORTED.ma1}inv(Kt)         % aborted.
%------------------------------------------------------------%
% O -> T: mr1 = {me1.me2}inv(Ko)             % resolve:
% T -> O: mr2 = {ABORTED.ma1}inv(Kt)         % aborted?
% T -> O: mr2 = {me1.me2}inv(Kt)             % resolved.
%
% OR:
%
% R -> T: mr1 = {me1.me2}inv(Kr)             % resolve:
% T -> R: mr2 = {ABORTED.ma1}inv(Kt)         % aborted?
% T -> R: mr2 = {me1.me2}inv(Kt)             % resolved.
%------------------------------------------------------------%
%
%  Non-repudiation goals are added to the IF specification
%  at this point, but strong fairness has been added as a
%  HLPSL goal macro.
%
%------------------------------------------------------------%

role Originator (
    O,R,T       : agent,
    Ko,Kr,Kt    : public_key,
    Text        : message,
    ABORTED     : message,
    H           : function,
    SND,RCV     : channel (dy),
    SND_P,RCV_P : channel (nro),
    Abort,Resolve : message
) played_by O def =

  local
```

```
    No          : text (fresh),
    Nr          : text,
    H_Nr        : message,
    Me1         : message,
    Ma1,Me2     : message,
    State       : nat

  init
    State = 0

  transition

 1. State  = 0  /\ RCV(start) =|>
    State' = 2  /\ SND_P(O,R;{Ko.Kr.T.Text.H(No')}_inv(Ko))
                /\ Me1' = {Ko.Kr.T.Text.H(No')}_inv(Ko)

 2. State  = 2  /\ RCV_P(R,O;{Me1.H(Nr')}_inv(Kr)) =|>
    State' = 4  /\ SND_P(O,R;No)
                /\ Me2' = {Me1.H(Nr')}_inv(Kr)

 3. State  = 4  /\ RCV_P(R,O;Nr) =|>
    State' = 6
                /\ received(O,Me1.No.Me2.Nr,Me1.No.Me2.Nr)
/\ accepted(O)

 4. State  = 2  /\ RCV(Abort) =|>                % Abort:
    State' = 8  /\ SND({ABORTED.Me1}_inv(Ko))
                /\ Ma1' = {ABORTED.Me1}_inv(Ko)

 6. State  = 8  /\ RCV({Me1.{Me1.H(Nr)}_inv(Kr)}_inv(Kt)) =|>   % Resolved.
    State' = 10
                /\ received(O,Me1.No.Me2.Nr,Me1.No.Me2.Nr)
/\ accepted(O)

 7. State  = 8  /\ RCV({ABORTED.Ma1}_inv(Kt)) =|>              % Aborted.
    State' = 100
                /\ accepted(O)

 8. State  = 4  /\ RCV(Resolve) =|>                             % Resolve:
    State' = 12 /\ SND({Me1.Me2}_inv(Ko))

 9. State  = 6  /\ RCV(Resolve) =|>                             % Resolve:
    State' = 12 /\ SND({Me1.Me2}_inv(Ko))

10. State  = 12 /\ RCV({ABORTED.Ma1}_inv(Kt)) =|>              % Aborted.
    State' = 14
                /\ accepted(O)
```

```
11. State  = 12 /\ RCV({Me1.Me2}_inv(Kt)) =|>                  % Resolved.
    State' = 16
     /\ received(O,Me1.No.Me2.Nr,Me1.No.Me2.Nr)
                /\ accepted(O)




end role

%-------------------------------------------------------------%

role Responder (
    O,R,T       : agent,
    Ko,Kr,Kt    : public_key,
    Text        : message,
    ABORTED     : message,
    H           : function,
    SND,RCV     : channel (dy),
    SND_P,RCV_P : channel (nro),
    Abort,Resolve: message
) played_by R def =

  local
    No          : text,
    Nr          : text (fresh),
    Me1,Me2,Ma1 : message,
    State       : nat

  init
    State = 1

  transition

 1. State  = 1  /\ RCV_P(O,R;{Ko.Kr.T.Text.H(No')}_inv(Ko)) =|>
    State' = 3  /\ SND_P(R,O;{{Ko.Kr.T.Text.H(No')}_inv(Ko).H(Nr')}_inv(Kr))
                /\ Me1' = {Ko.Kr.T.Text.H(No')}_inv(Ko)
                /\ Me2' = {{Ko.Kr.T.Text.H(No')}_inv(Ko).H(Nr')}_inv(Kr)

 2. State  = 3  /\ RCV_P(O,R;No) =|>
    State' = 5  /\ SND_P(R,O;Nr)
                /\ received(R,Me1.No.Me2.Nr,Me1.No.Me2.Nr)
/\ accepted(R)


 9. State  = 3  /\ RCV(Resolve) =|>                             % Resolve:
```

```
    State' = 7  /\ SND({Me1.Me2}_inv(Kr))

 9. State  = 5  /\ RCV(Resolve) =|>                          % Resolve:
    State' = 7  /\ SND({Me1.Me2}_inv(Kr))


10. State  = 7  /\ RCV({ABORTED.Ma1}_inv(Kt)) =|>           % Aborted.
    State' = 9
                /\ accepted(R)

11. State  = 7  /\ RCV({Me1.Me2}_inv(Kt)) =|>               % Resolved.
    State' = 11
                /\ accepted(R)
/\ received(R,Me1.No.Me2.Nr,Me1.No.Me2.Nr)

end role

%-----------------------------------------------------------%

role TrustedThirdParty (
    T           : agent,
    Kt          : public_key,
    ABORTED     : message,
    H           : function,
    A_List      : message set,
    R_List      : message set,
    SND,RCV     : channel (dy),
    SND_P,RCV_P : channel (nro),
    Abort,Resolve: message
) played_by T def =

  local
    Me1,Me2,Ma1 : message,
    State       : nat,
    K           : public_key,
    Count       : nat

  init
    State = 10
 /\ Count = 0

  transition

% O -> T: ma1 = {ABORTED.me1}_inv(Ko)         % abort:
% T -> O: ma2 = {me1.me2}_inv(Kt)             % resolved?
% T -> O: ma2 = {ABORTED.ma1}_inv(Kt)         % aborted.
%
```

```
% O -> T: mr1 = {me1.me2}_inv(Ko)             % resolve:
% T -> O: mr2 = {ABORTED.ma1}_inv(Kt)         % aborted?
% T -> O: mr2 = {me1.me2}_inv(Kt)             % resolved.
%
% OR:
%
% R -> T: mr1 = {me1.me2}_inv(Kr)             % resolve:
% T -> R: mr2 = {ABORTED.ma1}_inv(Kt)         % aborted?
% T -> R: mr2 = {me1.me2}_inv(Kt)             % resolved.

 1. State  = 10 /\ RCV({ABORTED.Me1'}_inv(K')) /\ in(Me1',R_List) =|>
    State' = 11 /\ SND({Me1'.Me2}_inv(Kt))
     /\ R_List' = cons(Me1',R_List)

 2. State  = 10 /\ not(in(Me1',R_List)) /\ RCV({ABORTED.Me1'}_inv(K')) =|>
    State' = 11 /\ SND({ABORTED.{ABORTED.Me1'}_inv(K')}_inv(Kt))
                /\ A_List' = cons(Me1',A_List)

 3. State  = 10 /\ RCV({Me1'.Me2'}_inv(K')) /\ in(Me1',A_List) =|>
    State' = 11 /\ SND({ABORTED.Ma1}_inv(Kt))

 4. State  = 10 /\ RCV({Me1'.Me2'}_inv(K')) /\ not(in(Me1',A_List)) =|>
    State' = 11 /\ SND({Me1'.Me2'}_inv(Kt))
     /\ R_List' = cons(Me1',R_List)



 5. State  = 11 /\ RCV({ABORTED.Me1'}_inv(K')) /\ in(Me1',R_List) =|>
    State' = 12 /\ SND({Me1'.Me2}_inv(Kt))
     /\ R_List' = cons(Me1',R_List)

 6. State  = 11 /\ RCV({ABORTED.Me1'}_inv(K')) /\ not(in(Me1',R_List)) =|>
    State' = 12 /\ SND({ABORTED.{ABORTED.Me1'}_inv(K')}_inv(Kt))
                /\ A_List' = cons(Me1',A_List)

 7. State  = 11 /\ RCV({Me1'.Me2'}_inv(K')) /\ in(Me1',A_List) =|>
    State' = 12 /\ SND({ABORTED.Ma1}_inv(Kt))

 8. State  = 11 /\ RCV({Me1'.Me2'}_inv(K')) /\ not(in(Me1',A_List)) =|>
    State' = 12 /\ SND({Me1'.Me2'}_inv(Kt))
     /\ R_List' = cons(Me1',R_List)


end role

%-------------------------------------------------------------%
```

```
role Session (
    O,R,T       : agent,
    Ko,Kr,Kt    : public_key,
    Text        : message,
    ABORTED     : message,
    H           : function,
    A_List      : message set,
    R_List      : message set,
    SND,RCV     : channel (dy),
    SND_P,RCV_P : channel (nro),
    Abort, Resolve: message
) def =

  composition
    Originator(O,R,T,Ko,Kr,Kt,Text,ABORTED,H,SND,RCV,
               SND_P,RCV_P,Abort,Resolve)
 /\ Responder(O,R,T,Ko,Kr,Kt,Text,ABORTED,H,SND,RCV,
              SND_P,RCV_P,Abort,Resolve)
 /\ TrustedThirdParty(T,Kt,ABORTED,H,A_List,R_List,SND,
                      RCV,SND_P,RCV_P,Abort,Resolve)

end role

%-------------------------------------------------------------%

role Environment() def =

  const
    o,r,t       : agent,
    ko,kr,kt    : public_key,
    the_text    : message,
    aborted     : message,
    h           : function,
    snd,rcv     : channel (dy),
    snd_p,rcv_p : channel (nro),
    abort       : message,
    resolve     : message

  knowledge(i) = {i,r,t,ko,kr,kt,aborted,h,abort,resolve}

  composition
    Session(o,r,t,ko,kr,kt,the_text,aborted,h,a_list,
            r_list,snd,rcv,snd_p,rcv_p,abort,resolve)

end role

%-------------------------------------------------------------%
```

```
goal

  strong fairness Originator, Me1.No.Me2.Nr, Responder, Me1.No.Me2.Nr

end goal

%-------------------------------------------------------------%

Environment()

%-------------------------------------------------------------%
```

B.3 The Internet Open Trading Protocol (IOTP)

```
%------------------------------------------------------------%
% IOTP Protocol - Purchase Transaction             18/10/04
%------------------------------------------------------------%
% Scenario:
%
%   Authenticion exchange
%   Offer exchange
%   Payment exchange
%------------------------------------------------------------%
% Authentication Exchange
%
%   C -> M: Offer
%   M -> C: Algorithm.Challenge.Trading_Role_Info_Request
%   C => M: Auth_Response.C
%   M -> C: Status
%------------------------------------------------------------%
% Offer Exchange
%
%   C -> M: Offer
%   M -> C: {C.M.P.Order.Payment}_inv(KeyM)
%------------------------------------------------------------%
% Payment Exchange
%
%   C -> M: Offer
%   M -> C: BrandList
%   C -> M: Selection
%   M -> C: {Payment.M.P}_inv(KeyM)
%   C -> P: {Status.{Payment.M.P}_inv(KeyM)}_inv(KeyC)
%   C <> P: ...
%   P -> C: Status.{Pay_Receipt.{Payment.M.P}_inv(KeyM)}_inv(KeyP)
%------------------------------------------------------------%
% Notes
%
% - Offer is a request message. E.g. HTTP
% - PKI is modelled as all roles possessing the other roles
%   public keys
%------------------------------------------------------------%
% Goals
%
%   P authenticates C on Payment
%
%------------------------------------------------------------%

role Customer (
    C,M,P                    : agent,
```

```
    Offer                     : message,
    Status                    : message,
    Trading_Role_Info_Request : message,
    KeyC,KeyM,KeyP            : public_key,
    SND,RCV                   : channel (dy),
    SND_A                     : channel (authentic)
) played_by C def =

  local
    State         : nat,
    Challenge     : text,
    Algorithm     : text,
    Order         : text,
    Payment       : text,
    BrandList     : text,
    Pay_Receipt   : text,
    Auth_Response : text (fresh),
    Selection     : text (fresh)

  init
    State = 0

  knowledge(C) = {inv(KeyC)}

  transition

 1. State  = 0  /\ RCV(start) =|>
    State' = 1  /\ SND(Offer)

 2. State  = 1  /\ RCV(Algorithm'.Challenge'.Trading_Role_Info_Request) =|>
    State' = 2  /\ SND_A(C,M;Auth_Response'.C)

 3. State  = 2  /\ RCV(Status) =|>
    State' = 3  /\ SND(Offer)

 4. State  = 3  /\ RCV({C.M.P.Order'.Payment'}__inv(KeyM)) =|>
    State' = 4  /\ SND(Offer)

 5. State  = 4  /\ RCV(BrandList') =|>
    State' = 5  /\ SND(Selection')

 6. State  = 5  /\ RCV({Payment.M.P}__inv(KeyM)) =|>
    State' = 6  /\ SND(Status.{Payment.M.P}__inv(KeyM))    %% to P
                /\ witness(C,P,payment1,Payment)

 7. State  = 6  /\ RCV(Status.Pay_Receipt'.{Payment.M.P}__inv(KeyM)) =|>
    State' = 7
```

```
end role

%-----------------------------------------------------------%

role Merchant (
    C,M,P                     : agent,
    Offer                     : message,
    Status                    : message,
    Trading_Role_Info_Request : message,
    KeyC,KeyM,KeyP            : public_key,
    SND,RCV                   : channel (dy),
    RCV_A                     : channel (authentic)
) played_by M def =

  local
    State         : nat,
    Challenge     : text (fresh),
    Algorithm     : text (fresh),
    Order         : text (fresh),
    Payment       : text (fresh),
    BrandList     : text (fresh),
    Auth_Response : text,
    Selection     : text

  init
    State = 0

  knowledge(M) = {inv(KeyM)}

  transition

 1. State  = 0 /\ RCV(Offer) =|>
    State' = 1 /\ SND(Algorithm'.Challenge'.Trading_Role_Info_Request)

 2. State  = 1 /\ RCV_A(C,M;Auth_Response'.C) =|>
    State' = 2 /\ SND(Status)

 3. State  = 2 /\ RCV(Offer) =|>
    State' = 3 /\ SND({C.M.P.Order'.Payment'}__inv(KeyM))

 4. State  = 3 /\ RCV(Offer) =|>
    State' = 4 /\ SND(BrandList')

 5. State  = 4 /\ RCV(Selection') =|>
    State' = 5 /\ SND({Payment.M.P}__inv(KeyM))
```

```
end role

%-----------------------------------------------------------%

role Payment_Handler (
    C,M,P          : agent,
    Offer          : message,
    Status         : message,
    KeyC,KeyM,KeyP : public_key,
    SND,RCV        : channel (dy)
) played_by P def =

  local
    State       : nat,
    Payment     : text,
    Pay_Receipt : text (fresh)

   init
     State = 0

   knowledge(P) = {inv(KeyP)}

   transition

 1. State  = 0 /\ RCV(Status.{Payment'.M.P}__inv(KeyM)) =|>
    State' = 1 /\ SND(Status.{Pay_Receipt'.{Payment'.M.P}__inv(KeyM)}__inv(KeyP))
               /\ request(C,P,payment1,Payment')

 2. State  = 1 /\ RCV(Status.{Payment'.M.P}__inv(KeyM)) =|>
    State' = 2 /\ SND(P,C;Status.Pay_Receipt'.{Payment'.M.P}__inv(KeyM))
               /\ request(C,P,payment1,Payment')

end role

%-----------------------------------------------------------%

role Session (
    C,M,P                     : agent,
    Offer                     : message,
    Status                    : message,
    Trading_Role_Info_Request : message,
    KeyC,KeyM,KeyP            : public_key,
    SND,RCV                   : channel (dy),
    SND_A                     : channel (authentic),
    RCV_A                     : channel (authentic)
) def =
```

```
  composition
    Customer(C,M,P,Offer,Status,Trading_Role_Info_Request,KeyC,KeyM,KeyP,SND,RCV,SND_A)
 /\ Merchant(C,M,P,Offer,Status,Trading_Role_Info_Request,KeyC,KeyM,KeyP,SND,RCV,RCV_A)
 /\ Payment_Handler(C,M,P,Offer,Status,KeyC,KeyM,KeyP,SND,RCV)

end role

%-----------------------------------------------------------%

role Environment() def =

  const
    c,m,p                     : agent,
    offer,status              : message,
    trading_role_info_request : message,
    keyC,keyM,keyP            : public_key,
    snd,rcv                   : channel (dy),
    snd_a,rcv_a               : channel (authentic)

  knowledge(i) = {i,c,m,p,offer,status,trading_role_info_request,keyC,keyM,keyP,snd,rcv,

  composition
    Session(c,m,p,offer,status,trading_role_info_request,keyC,keyM,keyP,snd,rcv,snd_a,rc

end role

%-----------------------------------------------------------%

goal

  Payment_Handler authenticates Customer on Payment

end goal

%-----------------------------------------------------------%

Environment()

%-----------------------------------------------------------%
```