Data independence in the model checking of security protocols

Philippa J. Broadfoot
University College

Thesis submitted for the degree of Doctor of Philosophy

Trinity Term, 2001

Data independence in the model checking of security protocols

Philippa J. Broadfoot, University College

Trinity Term, 2001

Abstract

The research presented in this thesis lies in the area of security protocol analysis, focusing mainly on confidentiality and authentication properties. The formal method used is CSP with the model checker FDR. This approach has proved to be very successful for modelling security protocols, especially when it comes to finding attacks (for example [Low95] and [LR97b]). However, since it can only check a small finite instance of a protocol model, this was incomplete as a method for proving correctness and limited when it came to analysing larger models. This thesis is concerned with enhancing this approach to enable a more complete analysis of security protocols.

The data independence techniques developed by Roscoe [Ros98a] allow models in which agents can perform an unbounded number of sequential runs, to be verified by means of a finite refinement check in FDR. This work is generalised here to cover a broad class of protocols and fully automated within the Casper compiler [Low98], thereby abstracting away the complexity of their implementation within the CSP models.

The problem of capturing parallelism within agents is addressed by the development of an alternative modelling approach, referred to as *internal agents*. Not only does this act as an excellent state space reduction strategy, but also enables higher degrees of parallelism to be captured within the CSP models than was previously feasible. A number of classes of protocols are derived for which this modelling technique captures arbitrary parallelism amongst the internal agents (still within the confines of a finite refinement check). The models are also extended to enable the construction of signal events for internal agents, thereby allowing these techniques to be applied to agents that are part of specifications. Further avenues benefiting from this work (for example, the ability to verify arbitrary parallelism among all agents) are discussed.

State space explosion is another common phenomenon in model checking approaches. A number of effective state space reduction strategies are developed within this framework.

Acknowledgements

I am greatly indebted to my supervisor Bill Roscoe for his guidance, wisdom and support he has provided me throughout the course of my M.Sc and D.Phil, both academically and otherwise. I would also like to extend my gratitude and thanks to Gavin Lowe, who has provided much insight and guidance, as well as countless discussions on the subject and proof reading this work. Additional thanks are to Michael Goldsmith for proof reading chapters of this thesis and David Walker for being a source of encouragement and support. A very special thanks goes to Heather Hamill for being the great friend that she is and for all the encouragement and laughs we have shared throughout my time here in Oxford. I would like to thank everyone in the Computing Laboratory, Linacre College and University College who have provided such a supportive and enjoyable environment to work and live in. Thanks are also due to EPSRC, DERA and the US Offices of Naval Research for funding this work. I thank Mike Reed and Catherine Meadows for acting as internal and external examiners respectively.

I could not have come this far without the continuous love, support and unwaivering encouragement from my parents Guy and Glen, my sister Samantha and my fiancé Brett. For this, I am greatly thankful to them. When I thought that life could not be more blessed with my wonderful family, I was proved wrong when Brett came into my life. I thank him for everything he has done and been throughout this last year.

Contents

1 Introduction **1**

1.1 Introducing security protocols 2

1.1.1 Why are security protocols difficult to analyse? 5

1.2 The problems this thesis addresses 7

1.3 Existing methods 9

1.3.1 BAN logic 9

1.3.2 General purpose model checkers 12

1.3.3 NRL Protocol Analyzer 13

1.3.4 Theorem proving approach: By induction 14

1.3.5 The spi-calculus 15

1.3.6 Rank functions 16

1.3.7 Strand Spaces 17

1.3.8 Thresholds and reduction techniques 19

1.4 Thesis overview 20

2 Security protocol models in CSP **22**

2.1 Introducing CSP and FDR 22

2.1.1 Basic principles and notation 23

2.1.2 The traces model 26

2.1.3 FDR model checker 27

2.2 Modelling security protocols in CSP 28

2.2.1 The CSP protocol model 29
2.2.2 Specification and verification of protocol goals 36
2.3 Introducing Casper 39
2.3.1 Casper input script 39
2.3.2 From Casper input scripts to interpreting the results . 46

3 Data independence techniques 48
3.1 Introducing data independence 49
3.2 Collapsing functions 50
3.2.1 Underlying Principles 50
3.2.2 Positive deductive systems 55
3.3 Application to CSP protocol models 56
3.4 Specification considerations 60
3.5 Summary 62

4 Automating data independence 64
4.1 Extending CSP protocol models 65
4.1.1 Trustworthy processes 66
4.1.2 Manager processes 68
4.1.3 Intruder process 72
4.1.4 Specification processes 72
4.2 State space reduction strategies 75
4.2.1 Strategy 1: Optimising Casper-generated scripts 75
4.2.2 Strategy 2: Stale knowledge 77
4.3 Automating data independence techniques 80
4.3.1 Extending Casper input script 80
4.4 Example: Analysing the Otway-Rees protocol 83
4.4.1 System being verified 84
4.4.2 Resulting analysis 84
4.5 Discussion 87

5 Internal agents **89**

5.1 Internal agents 91

5.1.1 Extending intruder's deductive system 93

5.1.2 Constructing internal agents 95

5.1.3 Example: Applying our internalisation strategy 96

5.2 Introducing false attacks 98

5.2.1 External agents versus internal agents 101

5.2.2 Supervisor processes 102

5.3 Extending CSP protocol models and Casper 105

5.3.1 Extending the CSP protocol models 105

5.3.2 Extending Casper 108

5.4 Some examples: Internal servers 110

5.4.1 Example 1: The Otway-Rees protocol 110

5.4.2 Example 2: The Yahalom protocol 114

5.5 Signal events for internal agents 115

5.5.1 Constructing signal events for internal agents 115

5.5.2 A basic requirement in practice 118

5.6 Example: Specifications with internal agents 118

5.7 Discussion 121

6 Capturing parallel attacks **123**

6.1 Just-in-time principle 126

6.2 Constructing a reduced protocol model 132

6.2.1 Dummy values 132

6.2.2 Constructing an equivalent reduced model 134

6.3 Factorisability of internal agents 139

6.4 Bounding the intruder's appetite 145

6.5 Example: Capturing parallel attacks 159

6.5.1 Protocol description 159

6.5.2 The resulting CSP protocol model 161

6.6 Discussion 165

6.6.1 Further research 165

7 Discussion **169**

7.1 Conclusion 169

7.2 Comparison with other existing approaches 171

7.3 Future Work 174

Bibliography **176**

List of Figures

2.1 Overview of the network in the CSP protocol models. 30

2.2 Casper - Compiler for the Analysis of Security Protocols. . . . 40

2.3 Casper script for the Yahalom protocol defined in Example 2.1. 41

4.1 Network in CSP models that incorporate the data independence techniques, using the NSPK protocol as the example. . 66

4.2 Casper input script for the adapted NSPK protocol example, incorporating the data independence techniques. 81

5.1 Supervisor process for internal agent A. 103

6.1 Applying the just-in-time property. 128

6.2 Violating the just-in-time property. 130

6.3 *Factorisable* internal server S in Example 6.4. 140

6.4 *Non-factorisable* internal server S in Example 6.5. 142

6.5 Performing the 3 factored runs of S in Example 6.7. 150

Chapter 1

Introduction

Establishing secure communications between computer systems through a public medium such as the Internet and being able to *prove* that these communications are indeed secure, is not only a growing area in academic research, but also one of great importance in society today.

Individual people as well as businesses are increasingly relying on the interaction between computer systems over large distances to perform a broad spectrum of confidential data transactions, for example, e-commerce, home banking and accessing company systems remotely to view (and potentially transfer) highly confidential information. It is a fast and effective means of data transmission. No need to rearrange your busy day to get to the bank on time. No need to travel back to the office on a Sunday to pick up those confidential files you needed to work on and left behind. No need for companies to wait for the shipping of large amounts of confidential documentation between one another.

However, the usefulness and therefore success of this form of communication relies on a very important factor (other than the naïveté of its users): The ability to ensure some level of security. To believe that no one is interested in accessing secret government transactions, confidential business details or banking details is to be naïve. As so many friends have said to me when discussing this issue: "But surely no-one would be interested in accessing my little student account?". The fact that millions of pounds disappear every year due to Internet fraud and that systems believed to be inaccessible to unauthorised users are brought down by hackers, should give us some indication that one cannot necessarily assume all these computer interactions are trustworthy and secure.

Establishing security in this context can broadly be divided into two main areas. The first, *cryptography*, concerns the actual encryption algorithms used to prevent a hacker from being able to decrypt a piece of cyphertext and gaining access to either the plaintext or the encryption key. As Schneier so ably puts it in [Sch96], cryptography is *"the art and science of keeping messages secure"*. The second area is *cryptographic protocols*, also known as *security protocols*, that makes use of cryptography to establish a number of security properties such as confidentiality, authentication, integrity and nonrepudiation.

The research presented in this thesis lies in the area of security protocol analysis. We therefore continue this chapter by introducing security protocols and discussing the importance of their analysis. This is a growing area of research; many attempts have been made over the last decade to develop effective tools to model them and the security properties they claim to achieve. As pointed out by many researchers, this has proved to be an extremely difficult task despite the simplicity of the protocols being modelled. We introduce the problems we address and the aims of this thesis in Section 1.2. To put our work into perspective, we present an overview of some of the main established methods developed over the years in Section 1.3; in Chapter 7, we then discuss how the work presented in this thesis compares to those methods and more generally, the contributions made to this field. We give an overview of the contents of this thesis in Section 1.4.

1.1 Introducing security protocols

Suppose you have Internet banking facilities and can therefore access your account remotely from home. When you request the website from your bank's server system, how do you know it is the bank's genuine website, as opposed to a fake one? You could naïvely enter all your bank details with the belief that it will all transfer across the network securely encrypted, when in actual fact your information has been diverted to a malicious intruder unencrypted (or encrypted with a key known by the intruder!). These sorts of security problems (and many derivatives like them) are not the result of flawed cryptography; the problem is that cryptography alone does not always suffice to address security properties such as authentication (required for the simple example scenarios presented above). For these, we need security protocols.

A security protocol is a sequence of steps taken between two or more parties (we will refer to them as agents), using cryptography to establish security properties in a potentially hostile environment (for example, the Internet). The security properties we will mainly be focusing on are confidentiality and authentication. Confidentiality is concerned with whether data intended to be shared secretly by a select group of agents only, can be accessed (in an unencrypted form) by other agents outside this group (without irresponsible actions by legitimate agents). Authentication between two agents is concerned with agents being assured of the identity of other agents they are running the protocol with. Informally, an agent A authenticated to another agent B means that if B believes he has completed a run with A, then A was indeed running the protocol with B; an intruder cannot successfully fake the identity of A to B. The precise meaning and definition of authentication and how one should capture this property in the various modelling techniques, has been the cause of many debates and disagreements within the security community over the years. We present the definitions we use throughout this thesis in Chapter 2.

The class of protocols that we will be referring to throughout this thesis relies mainly on the following two cryptographic algorithms:

Symmetric-key cryptography: A message is encrypted and decrypted with the same key (or at least they are easily derivable from each other). Thus, if an agent A knows a message M encrypted by the key K (written M_K) and knows K, then A can decrypt M_K with K and learn M. Such cryptosystems suffer from the well known key-distribution problem, namely for two agents to communicate securely with one another, they must exchange a secret symmetric key beforehand.

Public-key cryptography: A message is decrypted with a different key than the one used for encryption. Thus, keys come in pairs, referred to as the *public-key* and *private-key* (or *secret-key*). This system relies on a secret key being devised that cannot be calculated by knowing the public key in any useful amount of time and vice versa. In 1976, Diffie and Hellman [DH76] brought cryptography to this new era through the introduction of special one-way functions known as trap-door functions[1]. These are one-way functions with the added property that, given an additional secret piece of data, one can easily compute the reverse

[1]Public-key cryptography was also developed earlier at GCHQ, but remained secret until 1998. Diffie and Hellman [DH76] derived this idea independently and were the first to publish it.

of such a function; without this secret, computing the reverse is intractable. In public-key cryptography, this extra information needed to easily reverse the encryption function applied using the public-key is the secret-key. Typically, the public-key of an agent A is, as its name suggests, publically available and therefore can be used to encrypt messages intended to be decryptable by A alone (who is assumed to be the sole owner of the corresponding secret key). This eliminates the key-distribution problem mentioned above. The most well known public-key cryptosystem is the RSA algorithm by Rivest, Shamir and Adleman [RSA78].

Note that this is a very simplistic description of these cryptographic mechanisms, with the sole intention of giving the reader a basic insight that suffices for the purposes of this thesis. We are only interested in the properties they provide, rather than any details of how they are implemented. For a more detailed introduction to the area of cryptography, see books such as *Applied Cryptography* by Schneier [Sch96] and *Codes and Cryptography* by Welsh [Wel98].

To illustrate what a typical protocol we will be using in this thesis looks like, consider the well known adapted Needham-Schroeder Public-Key (NSPK) protocol (shortened version), corrected by Lowe in [Low95], defined as follows:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B & : \{n_a,\ A\}_{PK(B)} \\ \text{Message 2.} & B \rightarrow A & : \{n_a,\ n_b,\ B\}_{PK(A)} \\ \text{Message 3.} & A \rightarrow B & : \{n_b\}_{PK(B)} \end{array}$$

This protocol consists of three simple steps between two agents A and B with the aim of authenticating them to one another. n_a and n_b represent fresh nonces introduced by A and B respectively. A nonce is a freshly generated piece of data that is not easily predicted or calculated in any way by other agents. A protocol frequently relies on the generation of fresh nonces by the participating agents in every run. This protocol makes use of public-key cryptography only; we write $PK(A)$ and $SK(A)$ to denote the public key and secret key of A respectively. For example, A sends a freshly introduced nonce n_a together with her identity and encrypted with B's public key, to B in message 1. Assuming that the public-key cryptosystem used is secure, this message can only be decrypted with B's secret key.

Frequently, protocols will make use of a trusted third party, known as a *server*. Servers are typically responsible for distributing secret keys, such as session keys between agents and the public keys of other agents upon request. The agents trust that the allocated server will behave honestly and will often each share a long term secret symmetric key with the server, established before the protocol run has started. As an example, consider the Yahalom protocol defined as follows:

Message 1. $A \rightarrow B \ : \ n_a$
Message 2. $B \rightarrow S \ : \ \{A,\ n_a,\ n_b\}_{SKey(B)}$
Message 3. $S \rightarrow A \ : \ \{B,\ k,\ n_a,\ n_b\}_{SKey(A)}$
Message 4. $S \rightarrow B \ : \ \{A,\ k\}_{SKey(B)}$
Message 5. $A \rightarrow B \ : \ \{n_b\}_k$

In this example, we have two agents A and B, together with a server S. The server is responsible for distributing a fresh session key to A and B on request. The aim is that, not only will A and B establish a secure communication this way, but they will also be authenticated to one another. There is not much point in sharing a secret symmetric key with someone, if you do not have some assurance that the person is indeed who you think they are. This protocol makes use of symmetric-key cryptography; we write $SKey(A)$ to denote a secret symmetric key shared by agent A and the server only and established before the protocol runs.

1.1.1 Why are security protocols difficult to analyse?

Many security protocols have been developed over the years. Many have been flawed, even under the assumption that the cryptography used is secure; some of these flaws were found years after they were first published and widely accepted as secure. One of the most well known of these was the attack found by Lowe [Low95] upon the original Needham-Schroeder Public-Key protocol [NS78], 17 years after it was published and "proved" by a number methods to be secure. These types of flaws are not caused by the cryptography used itself, but rather how it is used. The fact that protocols (such as the NSPK and Yahalom protocols introduced above) look so simple, has proven to be hugely deceptive when it comes to analysing them and proving their correctness.

There are many ways in which a malicious agent (referred to as an intruder) can manipulate agents within protocol runs, resulting in attacks that leave the participants none the wiser. For example, *replay attacks* (also referred to as *freshness flaws*) arise when agents running a protocol are unable to distinguish a stale message (old one used in a previous run) from a fresh one. The intruder attacks the protocol by replaying stale messages to honest agents. If successful, he is able to convince an honest agent A that she is running the protocol with another agent B, when in fact it is the intruder impersonating B. A well known attack in this category is the Denning Sacco attack [DS81] upon the Needham-Schroeder Symmetric-Key protocol [NS78]: The intruder is able to manipulate an honest agent into accepting a stale session key (that was generated by the server in an earlier run). This could cause vulnerabilities, for example, if the intruder learnt the stale key beforehand. Variables such as nonces and timestamps are introduced for the purposes of preventing this type of attack. Replay attacks can also arise in the case where a number of messages (or sub-components of messages) in a protocol description have the same form. The intruder could exploit this by replaying a message as a different one in the same run.

Other types of attacks include: *man-in-the-middle* attacks, where the intruder attacks the protocol by interposing himself between the communications of honest agents; *oracle* attacks, where the intruder uses honest agents to unwittingly decrypt messages that he can later use to attack the protocol; *interleaving* attacks, where the intruder engages in a number of runs interleaved, and sending messages received in one run as input to another. A classic attack that combines the interleaving and oracle approach, is Lowe's attack [Low95] upon the original Needham-Schroeder Public-Key protocol [NS78]. This list is by no means exhaustive. For further details on this, see [CJ96, Sch96, RSG$^+$01]. In addition, the intruder can use combinations of these strategies, making the prediction of how vulnerable a protocol is to them, even more difficult.

Despite their simple structure, modelling security protocols is a very difficult task. They are highly concurrent systems, since they involve (potentially unboundedly) many agents communicating with one another independently across a medium such as the Internet. In order to verify the broad spectrum of vulnerabilities thoroughly, one must be able to consider multiple runs, parallel runs and all possible interactions that can take place between agents within a hostile environment. Modelling such an environment in itself is a complex problem, since one needs to capture malicious intruders that can overhear and intercept messages, continuously deduce facts based on new information gathered and construct new messages based on current knowledge.

An important contribution in the early stages of this development was made by Dolev and Yao [DY83]. The authors introduced the idea of having the network controlled by an intruder component with the following abilities: i) overhear all messages flowing through the network, ii) intercept messages, iii) fake messages based on information currently stored in its memory (either by constructing new messages or simply replaying old ones), and iv) behave as a legitimate agent running the protocol. Essentially, the intruder is acting as an unreliable medium through which all communications between the participants of the protocol pass. Most of the formal analysis methods developed to date (including the methods presented in this thesis), incorporate this intruder model (or some variant derived closely from this one).

Reasoning about concurrent systems such as these are notoriously difficult. Apart from the complexity of developing them, one of the common problems encountered is state space explosion. Protocols frequently rely on values, such as keys and nonces, being fresh and unique for every protocol run. As the number of agents and cardinality of the sets of fresh values increases, so does the state space. This can be a limiting factor depending on the techniques being used for analysis.

Another facet of security protocols that adds to their complexity is determining what security properties they are supposed to ensure and how to capture them within the analysis approach. This has been the cause of much dispute over the years, resulting in many different notions and definitions for them. The definitions we use in this thesis are given in Chapter 2.

As Clark and Jacob so eloquently put it in [CJ96]: *"The current explosion in distributed system development and network usage means that there is a pressing need for a framework and tool support for the rigorous development and analysis of new security protocols. Although significant advances have been made in recent years, there is clearly some way to go!"*

1.2 The problems this thesis addresses

There are a number of approaches used today to analyse security protocols. We give an overview of the main ones in Section 1.3 of this chapter. However, most of them suffer from one or more of the following drawbacks: lack of tool support, state space explosion, inefficiency, restrictions upon the size of the systems modelled, lack of expressive power, high levels of expertise required, ambiguity or no guarantee of giving a result (non-terminating).

In this thesis, we seek to address a number of these shortfalls within the world of model checking. In particular, the framework we will be working in is the well established CSP approach to modelling security protocols, together with the FDR model checker.

Model checkers such as FDR have proved to be a very useful and effective tool for modelling security protocols. In particular, FDR has proved to be very successful in finding attacks on protocols (see for example [Low95] and [LR97b]). Unfortunately, FDR do not allow processes to be defined with infinite-ranged parameters. This means that one is restricted to verifying finite instances of security protocol models. In addition to this constraint, the size of these sets must remain relatively small to avoid the state space of the implementation exploding to an unmanageable size. The agents within the protocol would then take a *fresh* data value from its predefined set until there were no more values left within the set. Once an agent had used up all its data values, it would stop. One of the problems with this approach is that once the participating agents runs out of its corresponding data values taken from these finite sets, the model must terminate, thereby limiting all protocol implementations to a finite run. So, whilst this framework is excellent for finding attacks upon protocols, one is unable to draw conclusions to the security of the protocols when an attack has not been found.

There are three main goals we intend to achieve. The first is to address the problem of only being able to model a small finite system; this is a problem that is common to most model checking techniques. Our aim is to extend the CSP framework for modelling protocols (for which a detailed presentation can be found in [RSG$^+$01]) to allow a more complete analysis using the FDR model checker, than hitherto. In particular, we are interested in being able to reason about unbounded sequential agent runs as well as higher (and in some cases arbitrary) degrees of parallelism, still within the confines of a finite model. This work was motivated by the development of data independence techniques by Roscoe in [Ros98a] and their natural applicability to security protocols within the CSP framework.

Secondly, the problem of accessibility is addressed. Many tools developed for analysing security protocols require a high degree of expertise, making them inaccessible to many users who are solely interested in the development and analysis of the protocols (as opposed to the actual mechanics of the formal methods used). Developing protocol models within the CSP framework is no exception, especially when we extend them the models to incorporate the data independence techniques. Lowe [Low98] originally addressed this issue and developed the Casper compiler. Casper is a program that takes a more abstract description of a protocol and generates the corresponding CSP description; this resulting output file can directly be loaded into the FDR model checker and verified. We extend Casper to incorporate the new models we develop in this thesis, thereby making them more accessible to a broader class of non-specialist users.

The third goal is to develop a number of effective state space reduction strategies throughout the development of our work. State space explosion is another common phenomenon in model checking approaches. Since we are interested in being able to perform a more complete analysis using model checkers, we must ensure that we are able to control the state space growth and avoid unnecessary explosions.

1.3 Existing methods

Needham and Schroeder [NS78] were among the first to propose the idea of applying formal methods to reasoning about security protocols. In this section, we give an overview of some of the well known modelling approaches used today.

1.3.1 BAN logic

One of the first attempts at reasoning about the authentication of security protocols using a more formal approach was made by Burrows, Abadi and Needham [BAN89]. They propose a logic, now referred to as the BAN logic, to analyse the beliefs of the trustworthy agents running the protocols and establish how these beliefs are affected as a consequence of receiving new information (via a set of inference rules). Informally, their method of analysis is summarised as follows.

Given the description of some protocol (in the standard form presented in the literature), each message M is transformed into a logical formula, known as the *idealised* version of M. Examples of the constructs used in their logic include (as defined by the authors in [BAN89]):

$A \models X$ *A believes* X and therefore may act as though X is true.

$A \triangleleft X$ *A sees* X. A receives X in some message and can read it.

$A \mid\!\sim X$	*A once said* X. At some point, A sent X in some message. The time scale of when it was sent is not known, but it is known that A believed X when sent.
$A \stackrel{K}{\longleftrightarrow} B$	The agents A and B may use the shared key K to communicate with the assumption that K will never be discovered by any other participant other than A and B, or a third party trusted by A or B.

Each idealised message is then annotated with assertions that represent the state of the system after each protocol step. The first assertion (located before the first protocol message) represents the initial assumptions made about the system, for example, keys shared between agents. The final assertion represents the conclusions drawn with regards to the final state of the system. The next step is to derive the beliefs held by each participant in the protocol. This is achieved by applying a set of inference rules to the assumptions and assertions. Examples of such inference rules include: "If an agent A believes that the key K is shared with some other agent B ($A \models A \stackrel{K}{\longleftrightarrow} B$) and sees a message M encrypted under K ($A \triangleleft \{M\}_K$), then A believes that B once said M ($A \models B \mid\!\sim M$)", written more formally in [BAN89] as:

$$\frac{A \models A \stackrel{K}{\longleftrightarrow} B,\ A \triangleleft \{M\}_K}{A \models B \mid\!\sim M}$$

The aim is to prove that, given an initial set of assumptions of the protocol being analysed, certain formulas are satisfied in the conclusions (final set of assertions). The authors look at authentication properties only (no claims are made concerning secrecy) and derive formulas to capture different aspects of authentication, for example:

$$\begin{array}{rcl} A & \models & A \stackrel{K}{\longleftrightarrow} B \\ B & \models & A \stackrel{K}{\longleftrightarrow} B \end{array}$$

represents the fact that A and B both believe they share the secret key K with each other exclusively.

The BAN logic is a simple logic that attempts to analyse the beliefs of trustworthy agents running a protocol and thereby verify whether expected beliefs are indeed satisfied by the participants at the end of a protocol run. The authors do not attempt to give a formal definition of authentication or address issues concerning secrecy or knowledge leakage to malicious intruders. The BAN analysis requires a number of stringent assumptions to be made, including the assumption that participants are trustworthy (therefore behave according to the protocol rules only); this excludes the open Internet environment that is most commonly used today.

This lack of formality (or sufficient clarity concerning the precise aims of the logic) has been the cause of numerous misunderstandings with regards to the authors' intentions. An example illustrating this was Nessett's critiques in [Nes90], where he constructed a protocol that was clearly flawed and demonstrated how the BAN logic analysis gave it a clean bill of health. The flaw arose from the first message in the protocol consisting of a secret key K shared between two agents A and B and encrypted under the secret key of A. Under the assumption that everyone else knows the corresponding public key of A, K is divulged by the sending of this message. Burrows *et al.* responded to Nessett's criticism in [BAN90] by claiming that his initial assumption, that A believes K is a secret key shared only with B was inconsistent with A divulging K in public by sending the first message. Furthermore, the authors quoted the following from their original paper [BAN89] in their defence:

> *"...we do not consider errors introduced by concrete implementations of a protocol, errors such as deadlocks, or even inappropriate use of cryptosystems. Furthermore, while we allow for the possibility of hostile intruders, there is no attempt to deal with the authentication of an untrustworthy principle, nor detect weaknesses of encryption schemes or unauthorised release of secrets. We focus on the beliefs of trustworthy parties involved in the protocols and on the evolution of these beliefs as a consequence of communication."*

Probably the best known example of a protocol shown to be correct by a number of independent groups including the BAN logic, and then subsequently proved to be flawed is the Needham-Schroeder Public-Key protocol presented in [NS78]. The authentication attack was found 17 years later by Lowe [Low95].

Thus, while the BAN logic has provided useful insights into protocol analysis and raised a number of important problems to be addressed in this field, its simplicity does limit its usefulness when it comes to analysing protocols in a more realistic environment, for example, in the presence of malicious intruders. Furthermore, it is not always clear how one should interpret the results generated by the BAN logic; as demonstrated over the years (and mentioned briefly above), this approach has been misinterpreted and used in the wrong context, leading to misguided results on numerous occasions.

Various extensions to the BAN logic have been proposed to provide a richer framework, for example, the GNY logic by van Gong, Needham and Yahalom [GNY90] and that proposed by Syverson and Oorschot [SvO94].

1.3.2 General purpose model checkers

A number of general purpose model checkers have proved to be extremely effective at finding attacks upon security protocols, for example as demonstrated in [Low96a, LR97b, MCJ97, MMS97]. The method we will be using in this thesis falls into this category, using the model checker FDR (and the CSP process algebra); we introduce this approach in detail in Chapter 2. Examples of other model checkers in this area include Murϕ (by Mitchell *et al.* [MMS97]) and Brutus (by Marrero *et al.* [MCJ97]).

Typically, these model checkers work by exploring the complete state space of a given system and verifying whether certain specified conditions are satisfied in each state; when a state is found that does not satisfy the conditions, the search terminates and corresponding counter-examples are generated. In terms of the protocol models, these counter-examples represent potential attacks upon the protocol.

The advantage of model checking methods is that the analysis is fully automatic and does not require user interaction during the search. In addition, one can implement compilers that translate a high level description of a security protocol (in the form commonly found in the literature) to the input language of a given model checker, thereby automating the process of analysis even further. For example, the compiler Casper (by Lowe [Low98]) generates the corresponding CSP models for the FDR model checker (described in more detail in Chapter 2), and CAPSL [Mil97] is a Common Authentication Protocol Specification Language with the intention of translating high level descriptions of protocols into a number of existing tools (for example, the NRL Protocol Analyser [BMM99] and Athena [Mil00], both discussed below).

The disadvantage is that the state space being searched needs to be finite (in order to terminate) and relatively small, to fit resources. Therefore their use for *proving* the correctness of protocols is limited. Furthermore, the problem of state space explosion is well known in model checking approaches, although many reduction strategies have been developed over the years in each case. We discuss the developments made in the CSP/FDR approach in more detail throughout this thesis.

1.3.3 NRL Protocol Analyzer

A well-established approach for analysing security protocols is the NRL Protocol Analyzer developed by Meadows [Mea96b]. This is a special-purpose verification tool written in Prolog and designed specifically for analysing security protocols. It has successfully been used to find undiscovered flaws in protocols (see [Mea91, Mea92]) as well as the numerous flaws already documented in the literature.

A protocol is modelled as a set of transitions of state machines, where the state machines represent the participants, together with an intruder component (closely based on the Dolev and Yao model [DY83]). The analysis of a protocol works by the user declaring an insecure state, representing a state that the protocol should never be allowed to reach; for example, a state in which the intruder knows information that was supposed to remain secret. The analyzer then performs a backwards search from this state in order to find an initial state via the transition rules. If such a state is found, then the path represents a potential attack upon the protocol. Since no bounds are placed upon the number of concurrent runs, the original state space is infinite (if no attack exists). However, in many cases the user is able to reduce it to a finite search by applying a number of pruning techniques developed to prove that infinite sets of states are unreachable.

The advantage of this approach is that it can be used not only for finding attacks, but also in many cases, for proving the correctness of protocols for any degree of concurrency amongst the participants. It does, however, require a degree of user interaction to guide the search and reduce it to a finite state space (although many aspects of this are partially automated). Furthermore, as is the case with the model checking approaches, the problem of state space explosion does arise. A number of reduction techniques have been developed within the analyzer to help control this.

1.3.4 Theorem proving approach: By induction

Paulson [Pau98] developed an inductive approach for analysing security protocols using the theorem prover Isabelle. A protocol is inductively defined as a set of traces, representing the possible communications (events) between the participants of the protocol; *Says A B M* is an example of such a communication and means that *"A sends message M to B"*. The number of participants considered is infinite, thus allowing proofs to be constructed for the correctness of protocols of arbitrary size.

The intruder's abilities are modelled by several additional inference rules, capturing the faking of messages and fresh values being compromised; the latter is known as the *Oops* rule. The interception of messages is captured implicitly by the model without the need for additional rules. These rules make use of a number of special operators, including *parts*, *analz* and *synth*, each inductively defined as the least set closed under a given set of rules. For example, if H is a set of messages, then *analz H* is defined (in [Pau98]) to be the least set including H and closed under decryption by known keys. An example of such an inference rule is the following for *decryption*:

$$\frac{\mathit{Crypt}\,K\ M \in \mathit{analz}\ H \qquad K^{-1} \in \mathit{analz}\ H}{M \in \mathit{analz}\ H}$$

where $\mathit{Crypt}\,K\ M$ represents the encryption of message M under key K, and K^{-1} represents the decrypting key of K. Similarly, *synth H* is defined to be the least set that includes H, agent names and guessable numbers, and is closed under pairing, hashing and encryption. The complimentary encryption rule of that for decryption above, is defined as follows:

$$\frac{M \in \mathit{synth}\ H \qquad K \in H}{\mathit{Crypt}\,K\ M \in \mathit{synth}\ H}$$

Security properties are captured as predicates over the traces and are defined using the special operators mentioned above. The analysis consists of inductively proving that they hold over all the traces, using the theorem prover Isabelle.

The advantage of this approach is that one can analyse protocols of arbitrary size with regards to the number of concurrent runs performed by the participants. This is advantageous over most model checking techniques when it comes to proving the correctness of protocols. The disadvantage is that these proofs are not automatic and require a high level of user interaction and expertise to generate them. This means that their accessibility to a broad class of users interested in verifying security protocols is more limited than more automated approaches (for example, those based on model checking techniques).

1.3.5 The spi-calculus

The spi-calculus, developed by Abadi and Gordon [AG99], is an extension of the π-calculus [SW01] with cryptographic primitives, such as encryption and decryption, and provides another framework for modelling security protocols.

Similar to other process algebras, the π-calculus models systems as parallel processes that synchronise with one another by message-passing handshakes (instantaneous) through channels. In addition to this (and unlike the other algebras), the π-calculus allows channel names to created dynamically and to be passed along channels. For example, given some channel name c known only to process P, P can pass c to another process Q; the *scope* of knowledge of channel c has been extended to include Q. The process of extending the scope of restriction of a channel (the set of all processes that know the channel) is known as *scope extrusion.* Thus, processes in the π-calculus framework are said to be *mobile* because their abilities to communicate can change dynamically over time.

These scoping rules form the central idea for modelling security protocols, since they guarantee that a malicious intruder (or in fact any other part of the environment) cannot gain access to channels that have been declared as secret between two processes; the only way of accessing them is if they are passed explicitly. Furthermore, the scoping rules are used to model the introduction of fresh session keys, for example, a server issuing a fresh symmetric session key to be shared between two agents only. This can be modelled by the introduction of a new channel passed to the corresponding agents upon which only they are able to exchange messages.

Security properties are expressed in terms of equivalences. Two processes P and Q are said to be *testing equivalent* if, although they have a different internal structure, a third process cannot differentiate running in parallel with P from running in parallel with Q. In this case, P and Q are said to be indistinguishable. The authors use this notion of testing equivalence for reasoning about security properties, since the environment (or the third party) can be regarded as the intruder. By carefully constructing the processes, they prove that the intruder is unable to gain secret information. A similar use of equivalence testing is used for authentication.

The spi-calculus provides an elegant framework for modelling security protocols, especially with the powerful scoping rules provided by the π-calculus and consequently the dynamic representation of the protocol models. However, the proofs of their correctness are done by hand and are rather complex; this approach does lack the tool support necessary to make it more accessible to a more general audience.

1.3.6 Rank functions

Schneider [Sch98] developed a method for analysing security protocols using a theorem proving approach based on CSP traces; details of this approach can be found in [RSG$^+$01]. The CSP model used is very similar to the one we use, described in more details in Chapter 2. His approach is based on the notion *rank functions*. A rank function is a function that maps intruder facts and special signal events (used to capture secrecy and authentication properties) to integers in the range of $\{0, 1\}$.

Briefly, the idea is that secrecy and authentication properties are verified by analysing the conditions under which the intruder can gain knowledge of certain facts. In the case of secrecy, the facts considered are clearly the ones that are expected to remain secret. In the case of the authentication of an agent A to some other agent B, the concern lies in ensuring that some fact (responsible for signalling the commitment of B) can only occur within the network if it has been preceded by some corresponding fact (responsible for signalling the running of A). The author models the notion of *"fact F_1 must be preceded by fact F_2"* by proving that if fact F_1 is blocked, then fact F_2 can never take place; in other words, the intruder can never gain access to F_2. The aim is then to show that all the facts generated in the network satisfy some property and that all the facts the intruder cannot (and should not) be able gain knowledge of, do not have this property. The aim of the verification is to identify such a property.

These properties are constructed by applying rank functions over the set of facts in the network; facts are classified according to whether they are available to the intruder or not. Those available to the intruder are assigned a rank value of 1, whilst those not available to him together with facts that are expected to remain secret are assigned the rank value 0. Informally, if one is able to construct such a rank function for a given protocol, then one can prove that the intruder is unable to gain knowledge of facts with a rank value of 0.

Finding suitable rank functions for a given protocol and performing the necessary verification proofs is not straightforward. Two tools are used to assist the analysis. The first is the theorem prover PVS (Prototype Verification System) [SOR93], extended with a number of more specialised theories concerning the rank function techniques. The second tool is the *RankAnalyser* that helps the user to construct a rank function for a given protocol.

Initially, the analysis was restricted to modelling finite size protocols only; this has been extended by Heather and Schneider [HS00] to allow more complete proofs to be constructed for some classes of protocols of arbitrary size. In this respect, this approach is advantageous over others that are limited to analysing finite models; however, it is not fully automated and does require a level of expertise. It is not always clear how to construct appropriate rank functions and what results have been established if no such functions exist.

1.3.7 Strand Spaces

Fábrega *et al.* developed an elegant method for reasoning about security protocols based on *Strand Spaces* [FHG98a, FHG98b]. A *strand* is a sequence of events that a participating agent may perform. In the case of trustworthy participants, these consist of *send* and *receive* events (represented by the symbols $+$ and $-$ respectively) according to the protocol definition. In addition to these events, an intruder strand will also have the typical ability as captured in the Dolev Yao intruder model discussed earlier; this includes, applying cryptographic operations, discarding messages, faking messages and constructing new messages based on the information he knows. As an example, the following strand represents the encryption of some message M with key K:

$$\langle -K,\ -M,\ +\{M\}_K \rangle$$

Similarly, the decryption strand is represented as follows:

$$\langle -K^{-1},\ -\{M\}_K,\ +M \rangle$$

A *strand space* is a set of strands that contains strands for the trustworthy participants as well as for the intruder. Strands can intertwine with one another; thus the sending of a message M from one strand can correspond to the receiving of M by another. A *bundle* is defined to be a subset of the strand space large enough to represent a complete protocol exchange and takes the form of a finite acyclic graph, where the edges express the causal dependencies of the nodes. These dependencies are captured by the following two causal relations:

$n_1 \rightarrow n_2$ means that n_1 sends a message M that is received by n_2.

$n_1 \Rightarrow n_2$ means that n_1 is an immediate causal predecessor of n_2 on the same strand.

A bundle C is complete in the sense that (i) if n_1 is a node in C and $term(n_1)$ is negative (receiver), then there is a unique n_2 such that $n_1 \rightarrow n_2$ is also in C, and (ii) if n_1 is a node in C and $n_2 \Rightarrow n_1$ exists, then $n_2 \Rightarrow n_1$ is also in the bundle C. Thus a bundle contains all the causal dependencies between the nodes for a given protocol execution, and forms the basic structure upon which correctness proofs (or attacks) are built. These proofs make use of a kind of induction principle (similar to Paulson's approach discussed earlier) based on a partial order relation ($\preceq_C$) and the construction of the set of $\preceq_C$-minimal elements for a given bundle C.

Within this framework, the authors prove the authentication of A to B sharing data items X by establishing that, whenever a bundle C contains a B strand using X, then C also contains a corresponding A strand that also uses X. In the case where the authentication requires a one-to-one relation between the runs of A and B, they prove that C contains a *unique* initiator strand using X. In the case of secrecy, the secrecy of some value v is established by showing that, for every bundle in the strand space, there does not exist a node (neither trustworthy nor malicious) that has v as an *unprotected* term (no cryptographic operations applied to v).

Song *et al.* [SBP01] extended this model and developed the tool Athena to perform these proofs automatically. Athena combines theorem proving and model checking approaches in such a way as to avoid the state space explosion problem. This is achieved by representing a set of protocol runs that differ only in the order of interleaving executions of participants (essentially a bundle) as one state and being able to reason about them simultaneously. Athena also takes advantage of a symbolic state representation and thereby can reason about arbitrary participants running concurrently. A proof is constructed (or an attack found) by performing a backward search from some initial state. This initial state typically represents a bundle that has been proved to satisfy certain properties. The search progresses by adding events performed by participants according to the precise *causal* relation in the given state. The search terminates once a proof has been established or an attack has been found. The search is not guaranteed to terminate; in this case, one can limit the number of participants running the protocol to a finite number, thereby ensuring that the search will terminate and generate a result.

This approach, especially with the tool Athena, has proved to be very effective for the analysis of security protocols.

1.3.8 Thresholds and reduction techniques

In [Low99], Lowe proves that under certain conditions satisfied by a protocol and its environment, if no secrecy attack exists upon some finite instance of the protocol, then none exists upon that protocol of arbitrary size. By finite instance, Lowe means one instance of each role participating in a protocol run. The theory was developed with the aim of making protocol analysis using model checkers more complete; in other words, being able to prove the correctness of protocols as well as finding attacks. The advantage of this approach is that model checkers are fully automated and can verify small instances of protocols very quickly.

Currently, the theory developed is only applicable to secrecy properties and is quite restrictive upon the class of protocols handled. However, it is believed that this work can be extended to address authentication properties, as well as relaxing some of the necessary conditions.

In [Sto99], Stoller looks at a similar question: How can one reduce the problem of proving the correctness of a protocol of arbitrary size to one of a small finite size? His approach involves calculating thresholds upon the protocols beyond which the result of the analysis remains unchanged. Once again, this would allow complete proofs to be generated by verifying a finite instance of the protocol. Stoller's approach addresses the issue of authentication as well as secrecy and the class of protocols for which such a threshold can be computed is broader than was the case in Lowe's work [Low99]. However, the thresholds that are currently yielded, even for simple protocols, result in a finite system whose size is infeasible to analyse using model checkers.

1.4 Thesis overview

In Chapter 2, we introduce the background material upon which this thesis is dependent. We commence by introducing the process algebra CSP (*Communicating Sequential Processes*) developed by C.A.R. Hoare [Hoa85], together with a brief overview of the FDR (*Failures/Divergences Refinement*) model checker for CSP. This is followed by an overview of how we model security protocols within this CSP framework; we introduce the security properties of interest and discuss how they are captured within the protocol models. Finally, we introduce the Casper compiler, developed by Lowe [Low98], that takes a high level description of a security protocol and generates the corresponding CSP model.

In Chapter 3, we present an overview of the techniques we use throughout this thesis. These techniques were developed by Roscoe [Ros98a] with the aim of enabling us to prove far more complete results regarding security protocols on model checkers than previously possible. Roscoe showed how techniques borrowed from data independence and related fields can be used to achieve the illusion that nodes can call upon an infinite supply of fresh values (for example, nonces and keys), even though the actual types used for them remain finite. It is thus possible to create models of protocols in which nodes do not have to stop after a small number of sequential runs and to claim that a finite-state run on a model checker has proved that a given protocol is free from attacks that could be constructed in the model used.

In Chapter 4, we present the design and implementation of these data independence techniques within our CSP protocol models. Whilst developing these techniques, it soon became apparent that these techniques cause a substantial increase in the state space required per protocol model, than was previously the case without them; we present a series of effective state space reduction strategies to address this problem. Although the application of these data independence techniques have proved to be successful (illustrated in [RB99] by a series of case studies), this process has also proved to be time-consuming, error-prone and requires a substantial level of expertise. To make our methods more accessible, they have been fully automated by integrating them into Casper.

In Chapter 5, we present an alternative approach to modelling agent processes within our CSP models, known as *internal agents*. This involves capturing the functionality of participating nodes within the intruder as a series of deductions, instead of modelling them as individual CSP processes.

The initial motivation for this work was as a state space reduction strategy; as we will demonstrate, this approach has indeed proved to be very effective for this purpose. However, a more recent motivation for this work is to enable our CSP models to handle arbitrary parallelism within a single agent identity.

In Chapter 6, we derive a set of conditions upon the intruder that prevents him from unreasonably storing fresh values without passing them on to participating agents; this problem arises through the internalisation strategy of agents introduced in Chapter 5. As introduced in Chapters 3 and 4, our data independence models rely on the fact that no components ever store an unbounded number of fresh values; when agents are modelled as internal, the intruder has this ability. This has created one of the most troublesome problems in our work. We seek to address this issue and derive a number of conditions upon the intruder that prevents him from storing an unbounded number of fresh values, without weakening his abilities. Furthermore, for a given class of protocols, we are able to use these bounds to justify the fact that we can capture attacks for any degree of parallelism within the internal agents, while still performing a finite refinement check.

In Chapter 7, we present the conclusions of this thesis and discuss how our results compare to the other existing methods (in particular, referring to those introduced earlier in Chapter 1). We also discuss future avenues that we are interested in pursuing as a result of this work.

Chapter 2

Security protocol models in CSP

In this chapter, we introduce the background material upon which this thesis is dependent. We commence by giving an overview of the CSP language that we will be using, together with a brief overview of the FDR model checker for CSP. We follow this by presenting an overview of how we model security protocols within this framework and introduce Casper, a compiler developed by Lowe in [Low98] that takes a high level description of a security protocol and generates the corresponding CSP model. More complete details concerning CSP can be found in [Hoa85] by C.A.R. Hoare and [Ros98b] by Roscoe. In addition to introducing CSP, the latter provides detailed descriptions concerning the theory and tools (in particular, FDR) developed by Roscoe that we rely on in this thesis, together with a broad range of applications.

2.1 Introducing CSP and FDR

CSP (*Communicating Sequential Processes*) is a process algebra, developed by C.A.R. Hoare [Hoa85], for describing and reasoning about systems that are composed of processes in parallel, interacting with one another through some form of communication. There is a broad range of such systems, for example communication protocols, distributed systems and railway networks. In this thesis, we will use CSP to describe and reason about security protocols, where the system is composed of a number of agent and intruder processes running some protocol and communicating with one another by sending and receiving messages; an overview of CSP protocol models will be given in the next section.

2.1.1 Basic principles and notation

A process is defined in terms of the communication it can perform, known as *events*; for example, the following process P_1 can communicate the event a followed by the event b and then becoming the process *STOP*:

$$P_1 \;\widehat{=}\; a \rightarrow b \rightarrow STOP$$

The processes *STOP* is one of two terminating processes defined in CSP; the other is the process *SKIP*. By terminating, we mean that they never communicate any events. The difference between the two lies in the *type* of termination they represent; *STOP* represents deadlock, whereas *SKIP* represents successful termination.

Events can be defined in terms of compound objects; in this case, we refer to the event name as a *channel* and specify the type of the values that can be passed upon it. To illustrate this, consider the following example process,

$$P_2 \;\widehat{=}\; in?x : T \rightarrow out.x \rightarrow STOP$$

P_2 can perform any $in.x$ event where the variable x ranges over the set T (represented by the ? operator); one can also write $in?x$ meaning that x ranges over the whole type defined for this data component upon the channel *in*. The channel *out* also has one data component of some type X; in this process, the event $out.v$ can be performed where v is the value bound to the variable x in the previous event. Since the *out* channel is used in this process to communicate the same value input upon the channel *in*, the set T must be a subset of X (otherwise x might be bound to a value v that is out of scope of the channel definition of *out*). So, the set of all possible events P_2 can perform, known as the *alphabet* of P_2, is defined as

$$\alpha P_2 \;\widehat{=}\; \{in.x,\ out.x \mid x \in T\}$$

The processes above define finite sequences of events, since they both end in the terminating state $STOP$. More useful processes are *recursive* ones, where the process eventually loops back to its initial state (or one of the named states of the process). For example,

$$P_3 \;\widehat{=}\; in?x : T \rightarrow out.x \rightarrow P_3$$

This process no longer terminates after two events as was previously the case with P_2; instead, P_3 can perform an infinite sequence of alternative *in* and *out* events. Furthermore, processes can be parameterised, where each instantiation represents a new state, for example,

$$P_4(X) \;\widehat{=}\; in?x : T \;\rightarrow\; out.x \;\rightarrow P_4(X \cup \{x\})$$

CSP operators

This section gives a very brief overview of the CSP operators we use. We only give informal descriptions of them here; more complete details and formal semantics can be found in [Ros98b], together with other operators that are omitted here.

$\Box$ is the *external* choice operator giving the environment the choice of events of two processes. Consider the following example,

$$P_5 \;\widehat{=}\; a \rightarrow P_5, \qquad P_6 \;\widehat{=}\; b \rightarrow P_6$$

$$P_7 \;\widehat{=}\; P_5 \;\Box\; P_6$$

P_7 offers its environment the choice of the first event in P_5 and that of P_6. If the environment chooses event a, then P_7 will behave according to P_5 (thereby only being able to perform a events); similarly in the case where the environment chooses b.

$\sqcap$ is the *internal* choice operator, allowing the *nondeterministic* choice of processes. By nondeterministic, we mean that the choice is no longer made by the environment, but rather by an internal decision within the processes themselves; the environment has no guarantee of which events will be offered. Consider the following example,

$$P_8 \;\widehat{=}\; P_5 \sqcap P_6$$

P_8 can internally choose to offer (or refuse!) either the event a or b. For example, if the environment wishes to perform a, it is possible for P_8 to decide only to offer b; in this case, a will be blocked.

Processes can be composed out of sub-processes placed in parallel, where they communicate by synchronising upon events they share in common (or as specified). This synchronisation is known as *handshake* synchronisation and there is no specified direction that information flows in; rather two processes P and Q will synchronise upon an event e precisely when they are both in a state willing to perform e. Any number of processes can participate in a single synchronisation (known as *multi-way synchronisation*).

The CSP parallel operators we use in our models are summarised as follows:

$P \underset{X}{\parallel} Q$ is the *generalised* parallel operator; the processes P and Q must synchronise upon all events in the set X.

$P\ {}_X\|_Y\ Q$ is the *alphabetised* parallel operator; the processes P and Q must synchronise upon all events in $X \cap Y$.

$P \mathbin{|||} Q$ is the *interleaving* operator. Unlike the two parallel operators above, the processes P and Q do not synchronise upon any events, even those they have in common; they run completely independently of one another.

For a given process, it is often desirable to be able to divide P's events into (i) those *visible* to the environment, and (ii) those internal to P and therefore *invisible* to P's environment. This is a common concept in systems that contain sub-components running in parallel; each component C would typically have internal behaviour that it can perform independently of the other components and then an interface through which C is prepared to communicate with them. If the internal behaviour of each component was visible to all the others, the design of the system would be very cluttered and more error-prone. In CSP, events are made invisible through the *hiding* operator described as follows:

$P \setminus E$ is the *hiding operator*, where P is a process and E is the set of events to be hidden from P's environment, thereby making them internal. For example,

$$P_9 \quad \widehat{=} \quad P_7 \setminus \{a\}$$

The visible events of P_7 are $\{a,\ b\}$, whereas the only visible event of P_9 is $\{b\}$.

A final construct we present in this section is the *renaming* of visible events:

$P[\![R]\!]$ behaves like the process P except that all the visible events are renamed according to the renaming relation R. For example,

$$P_{10} \;\widehat{=}\; P_7\,[\![a \leftarrow c,\ b \leftarrow d]\!]$$

P_{10} is a process that is defined to behave like the process P_7, except that the events a and b are replaced by the events c and d respectively. Thus, this is equivalent to the following process,

$$P_{11} \;\widehat{=}\; c \;\rightarrow\; P_{11} \;\Box\; d \;\rightarrow\; P_{11}$$

Renaming allows a higher level of abstraction in the design of systems and if used correctly, provides more clarity of the individual components and their composition as a whole. We will see in the next section, how renaming is a useful construct in our protocol models. In Chapter 4, we will also see how renaming can prove to be an intuitive approach towards improving the efficiency of our models.

2.1.2 The traces model

In this section, we briefly introduce the *traces* model defined by Roscoe (see [Ros98b] for example) that we use for modelling and reasoning about security protocols in this thesis.

A *trace* is a sequence of events; $traces(P)$ is the set containing all possible traces that P can perform (visible to the environment). For example, the only trace of the process $STOP$ is the empty trace (since no event can ever be performed) and:

$$traces(P_1) \;=\; \{\langle\rangle,\ \langle a\rangle,\ \langle a,\ b\rangle\}$$

In the case of a recursively defined process, the set of all traces may be infinite; for example, the traces of the following process,

$$P_{12} \;\widehat{=}\; a \;\rightarrow\; P_{12}$$

is the infinite set:

$$traces(P_{12}) \;=\; \{\langle\rangle,\ \langle a\rangle,\ \langle a, a\rangle,\ \langle a, a, a\rangle,\ \langle a, a, a, a\rangle,\ \ldots\}$$

for all possible sequences of a. For details concerning the formal rules and semantics for constructing the traces under the CSP operators, see [Ros98b].

The refinement ordering we use in the traces model is defined:

$$S \sqsubseteq_T I \quad \Leftrightarrow \quad traces(I) \subseteq traces(S)$$

Thus, a process I *trace-refines* a process S precisely when every trace of I is also a trace of S. In the *traces* model, this refinement ordering allows us to verify safety properties only. We are able to say that a process P does not perform any unwanted behaviour that is not specified by a given specification process *Spec*, by checking whether $Spec \sqsubseteq_T P$ is satisfied. However, this model does not allow us to capture liveness properties: stating whether a process actually performs specified behaviour. The reason is that, although the *traces* of a process tells us what the process is able to offer, it does not say anything about the events a process may refuse in each state. Roscoe's text describes two richer models, namely the *stable failures* and the *failures/divergences* models that address this issue and allow a broader spectrum of properties to be captured (including liveness properties).

For the purposes of the work presented in this thesis, the *traces* model suffices, since we will only be concerned with verifying safety properties.

2.1.3 FDR model checker

FDR (*Failures/Divergences Refinement*) is the model checker we use to automatically perform all the refinement checks in our CSP protocol models. This tool was developed and is a commercial product of Formal Systems Ltd. [For97]. In addition to refinement checking, it also performs deadlock, livelock and determinism verification.

Informally, to verify whether a process *Spec* is refined by another process *Impl*, FDR performs an exhaustive breadth first search through the state space of these two processes and checks whether every trace in *Impl* is also present in *Spec*. If a trace t is found in *Impl* that does not exist in *Spec*, the refinement check fails and FDR will give the trace t as a counter-example to the user. This is essential as it provides a concrete example of unwanted behaviour in *Impl* that violates the specification requirement *Spec*. When analysing the counter-example, there are two advantageous features that FDR possesses. Firstly, for a given counter-example with trace t, FDR allows the user to step through all the processes of the system at each level and see what sequence of events each of them have performed up to the last event in t. This is a very useful feature when analysing and debugging the system being checked.

The second advantage is that, because FDR performs a breadth first search, it will return (if any exists) a shortest counter-example. For many applications, this is extremely beneficial, since it avoids any extra noise and therefore often make the counter-example more intuitive and clearer to understand. This is certainly true in the area of security protocol analysis that we are concerned with, since the counter-examples represent attacks by a malicious intruder; in addition to clarity reasons, one wants to know what the minimum amount of work required by the intruder is, to break the security protocol in question.

Since FDR works by performing an exhaustive state space search, the system being verified must be finite.

2.2 Modelling security protocols in CSP

Security protocols are traditionally described in the literature as a series of messages between the various legitimate participants; at a slightly more abstract level, one could regard this as a number of processes in parallel sending and receiving messages between one another, as specified by the protocol description in question. Since CSP is a notation designed specifically for describing systems that are composed of processes in parallel and communicating with one another by synchronisations upon common events (reflecting the sending and receiving of messages), it is ideal for modelling security protocols. Furthermore, using the model checker FDR, we are able to perform any necessary analysis upon the protocol modelled. This method, and model checkers in general, have indeed proved to be extremely effective at finding attacks upon security protocols, for example as demonstrated in [Low96a, LR97b, MCJ97, MMS97].

In order to clarify the presentation of our CSP models and make them more intuitive, we will use the Yahalom protocol presented below in Example 2.1 throughout the rest of this chapter. Although this particular version has a well known attack found by Syverson [Syv94], it provides a useful example for the purposes of introducing our methods. Further on in this chapter, we will discuss how the protocol models are analysed and if attacks exist, how they are found and represented.

Example 2.1 *The version of the Yahalom protocol (closely based on the one suggested in [BAN89]) we refer to in this section is defined as follows:*

$$
\begin{array}{llll}
\textit{Message 1.} & A \rightarrow B & : & N_a \\
\textit{Message 2.} & B \rightarrow S & : & N_b,\ \{A, N_a\}_{SKey(B)} \\
\textit{Message 3.} & S \rightarrow A & : & N_b,\ \{B, K_{ab}, N_a\}_{SKey(A)} \\
\textit{Message 4.} & S \rightarrow B & : & N_b,\ \{A, K_{ab}, N_b\}_{SKey(B)} \\
\textit{Message 5.} & A \rightarrow B & : & \{N_b\}_{K_{ab}}
\end{array}
$$

where N_a and N_b are nonces freshly introduced by the agents A and B, and K_{ab} is a session key freshly introduced by the server S.

■

The only difference from the version in [BAN89] is the way in which the server communicates message 4 directly to B rather than using A as a messenger. It has been demonstrated by Hui and Lowe [HL01] that this type of re-direction has no effect on security.

In this section, we present an overview of how we model security protocols in CSP, using the Yahalom protocol as a running example. This will be followed by an overview of the types of security goals we are interested in verifying and how these are captured as appropriate specifications within our framework. A brief overview of how our CSP protocols are analysed through FDR follows and finally, we discuss some of the limitations of these methods, to place the aims of this thesis more in context. The application of this framework to security protocols was initiated and developed mainly by Gardiner, Goldsmith, Jackson, Lowe and Roscoe, as part of a series of DRA strategic research projects led by Ryan.

2.2.1 The CSP protocol model

The CSP protocol model consists of a number of *trustworthy participant* processes (one for each participant running the protocol) together with an *intruder* process, all running in parallel and interacting via a set of communication channels. In the case of a trustworthy agent process, these interactions will be determined by the protocol description in question as we will see in examples presented below. The intruder, on the other hand, has a more complex functionality and a broader scope of abilities, since his

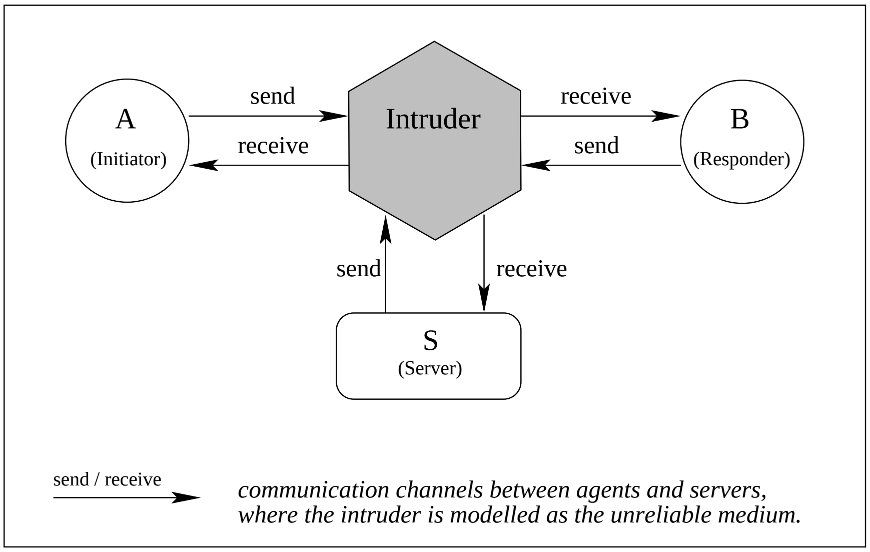

Figure 2.1: Overview of the network in the CSP protocol models.

task is to try and find security flaws by overhearing, intercepting and faking messages.

Typically, a protocol comprises a number of agents interacting with each other to establish the security goals specified by the protocol (for example, establishing a short-term session key). In addition, there is often a server process known to and trusted by all or most of the participating agents (for example, whose function is to provide fresh session keys). In the case of our Yahalom protocol example, there are three trustworthy roles that form part of a run: an *Initiator* agent (A), a *Responder* agent (B) and a *Server* role (S). As illustrated in Figure 2.1, they can *send* and *receive* messages in accordance with the protocol description, over whatever medium is specified; in our standard models, the intruder acts as the unreliable medium through which all communication flows, as discussed in more detail later on. For each agent (and similarly server) role, we construct a corresponding CSP process that accurately reflects their functionality as specified by the protocol. A protocol description, such as that given for the Yahalom protocol above, implicitly describes the role of each participant and carries the implication that whenever an agent has, *from its point of view*, executed all the communications implied by the protocol, then it can take whatever action is enabled by the protocol. It is relatively straightforward to derive CSP processes to represent them; indeed the tool Casper (described in Section 2.3) performs this task automatically. To illustrate how we achieve this, let's consider the *Initiator* agent A as an example. From the Yahalom protocol description, it is straightforward to identify the messages A sends and receives, the order in which they are performed and the expectations (if any) of the format and contents of received messages. In this case, A initiates a run of the protocol by sending a message 1 to some agent B, containing a fresh nonce that A introduces. Once A has sent this message, she expects to receive a message 3 of the form:

$$N_b,\ \{B, K_{ab}, N_a\}_{SKey(A)}$$

where N_a is bound to the same nonce A introduced in message 1 and B is bound to the agent identity for whom her message 1 was intended; A is happy to accept any values bound to the other variables. Once this corresponding message has been received, A then sends the final message 5 of the form:

$$\{N_b\}_{K_{ab}}$$

where the values bound to N_b and K_{ab} are the same as those bound to them in the message 3 that A received. More formally, we can capture this series of events for A by the following CSP process[1]:

$$\begin{array}{l}
\mathit{Initiator}(A, S, N_a) \mathrel{\widehat{=}} \\
\quad \Box\, B : \mathit{Agent} \bullet \\
\qquad \mathit{env} \,.\, A \,.\, B \;\rightarrow\; \mathit{send} \,.\, A \,.\, B \,.\, N_a \;\rightarrow \\
\qquad \Box\, K_{ab} : \mathit{SessionKey},\ N_b : \mathit{Nonce} \bullet \\
\qquad\quad \mathit{receive} \,.\, S \,.\, A \,.\, \langle N_b,\ \{B, K_{ab}, N_a\}_{SKey(A)} \rangle \;\rightarrow \\
\qquad\quad \mathit{send} \,.\, A \,.\, B \,.\, \{N_b\}_{K_{ab}} \;\rightarrow\; \mathit{SKIP}
\end{array}$$

where the parameters A, S and N_a represent the Initiator's identity, the server's identity and A's fresh nonce respectively.

In the CSP process above, we see that there are three channels of communication. The initial event is performed using the channel *env* and represents

[1] As a slight simplification this code omits the option of a node that is waiting for a message from another agent timing out, thereby terminating the run.

the choice (for example, by its local environment) of which agent identity A wishes to interact with (assigned to B). The *env* channel is used for communication between an agent (or server) and its environment, and therefore does not flow through the intruder. The agent identity is taken from some defined set *Agent* and there is nothing preventing A from requesting to interact with herself (for example, A may have processes running in different locations and need to establish secure connections between them). In CSP terms, we model this as an external choice ranging over the set *Agent* as follows:

$$\square\, B : \mathit{Agent} \;\bullet\; \mathit{env} \,.\, A \,.\, B \;\rightarrow\; \ldots$$

The other two communication channels are the *send* and *receive* channels that we briefly introduced above. These represent the main channels that all the agents and servers use to communicate with each other. Hence an event of the form $\mathit{send}.A.B.M$, represents the *sending* of message M by agent A intended for agent B. Similarly, an event of the form $\mathit{receive}.A.B.M$, represents the *receiving* of message M by agent B, supposedly sent by A.

At the point where A is ready to receive a corresponding message 3, we use the external choice ranging over all possible session keys and nonces (represented the sets *SessionKey* and *Nonce* respectively); this reflects the fact that A is prepared to receive any key and nonce bound to K_{ab} and N_b respectively. It is worth noting that this does not only apply to single types such as session keys and nonces, but can also be used for complete messages structures. This is particularly useful when an agent receives a message or (part of a message) intended to be forwarded, to another agent; for example, consider the following sequence of messages:

$$\begin{array}{ll} \text{Message } n. & S \rightarrow A \;:\; M,\, M' \\ \text{Message } n+1. & A \rightarrow B \;:\; M' \end{array}$$

where M and M' are two encrypted messages and A simply forwards M' onto B without being able to decrypt or change it. In this case, we would want to model A as being able to receive an arbitrary message bound to M' in messages n. We do indeed capture this in our CSP models by applying the external choice operator over all possible messages in the form of M' in the same way we did for the nonces and keys above.

The CSP process defined above for the *Initiator* agent was presented at a higher level than that used within the implementation code, providing a more intuitive introduction to how we construct our models. However, the actual CSP protocol implementations make use of abstract data types to represent the messages and define functions over these types to capture constructs such as encryption and hashing. For example, the encrypted message construct seen in the second part of message 3 in our Yahalom protocol example taking the form

$$\{B, K_{ab}, N_a\}_{SKey(A)}$$

would be captured in our actual CSP models by the following construct:

```
Encrypt.(SKey.A, Sq.<B, Kab, Na>)
```

where `Encrypt`, `SKey` and `Sq` are all defined as formal constructors in such a way that our model treats them as encryption, a symmetric key and a sequence respectively. All the messages passed around in the protocol implementation are drawn from an abstract, constructed data type that allows us to treat operations such as encryption in a symbolic way.

Within this framework, the abilities of agent processes to encrypt and decrypt messages are implicit in the way they create messages and understand the messages they receive. The fact that various states will only accept messages of a given form, sometimes only when containing a specific nonce, effectively gives a way of coding in the conditions for progress in the protocol. To all intents and purposes, such entries in the protocol represent an equality test (between the nonce expected and that actually received) with progress only being possible when the test succeeds.

Modelling the intruder

Our intruder model is to all intents and purposes the standard Dolev/Yao [DY83] one and is conceptually simple. He is able to act as other agents, which may or may not behave in a trustworthy way; overhear all messages that pass between the trustworthy agents; prevent messages from reaching their intended recipients; and finally, fake messages to any agent, purporting to be from any other. He is subject only to what the rules built into the crypto-system in use allow him to create based on his initial knowledge and subsequent listening.

We make the so called *perfect encryption assumption*: The intruder is able to produce an encrypted message only if he knows the encrypting key and is able to decrypt messages only if he knows the decrypting key; he cannot, for example, *guess* unknown keys or perform dictionary attacks. The rules determining how the intruder can deduce information and construct messages are defined by a set of deductions. Deductions are pairs (X, f) where, if the intruder knows every fact in the set X, then he can also generate f (also written as $X \vdash f$). The basic deductions one would commonly expect the intruder to have would be for encrypting and decrypting messages, as follows:

$$M,\ k \ \vdash\ \{M\}_k \tag{2.1}$$
$$\{M\}_k,\ k^{-1} \ \vdash\ M \tag{2.2}$$

where (2.1) represents the encryption of a message M by key k and (2.2) represents the decryption of the encrypted message $\{M\}_k$ by the corresponding decrypting key k^{-1} (of k). Depending on the form of encryption used, the intruder can be given further deduction rules, including deductions to handle Vernam encryption and decryption, hashing and function applications. Details concerning these deduction rules can be found in [Low98]. We refer to the complete set of deductions within a protocol model as the intruder's *deductive system*.

The intruder process is then constructed by applying a suitable renaming to one with the following behaviour. It is initialised so that it has whatever information X we deem appropriate. This typically consists of all *public* information, whatever information is necessary to allow the intruder to function as the other agents it can play the role of, and some nonces that are distinct from those that the trustworthy agents may invent for themselves.

$$\begin{array}{rcl} Spy(X) & \mathrel{\widehat{=}} & learn?x \rightarrow Spy(Close(X \cup \{x\})) \\ & & \Box\ say?x : X \rightarrow Spy(X) \end{array}$$

where $Close(X)$ is a function that produces all facts derivable from the set X under the chosen deductive system.

In order that this intruder process can even potentially be run on FDR, it is necessary that the sets of facts over which it ranges and deductions that it uses are finite. Thus the various data types making up the messages such as agent identities, keys and nonces need to range over finite sets. In practice these have to be kept very small: no more than a few values each, if the sizes of the sets of facts and deductions are not going to grow beyond the range that can be reasonably handled by FDR. In fact, this definition of the intruder cannot be run on present versions of FDR because the compilation regime it uses would insist on computing all its approximately 2^N states, where N is the number of facts that the parameter-sets are built from.

Roscoe and Goldsmith [RG97] developed a more efficient representation, namely a parallel composition of one two-state process for each learnable fact with deductions being carried out by communication over a special channel, known as *infer*. They showed that this resulting process is equivalent to the simple process above. Furthermore, they devised a partial order state exploration strategy (known as *chase*) that could be applied to these deductions without weakening the intruder's abilities. This is based on the fact that once a deduction is enabled in a state, it can never be disabled; in a given state, every ordering of deductions leads to the same result. Hence, it suffices to consider a single interleaving of the deductions in each state, rather than having to explore all of them. This is essentially what *chase* does: $chase(P)$ behaves like the process P unless it reaches a state where it can perform a τ action; in this case, it executes τ actions until there are no more available. The other interleavings are not explored. This observation made by Roscoe and Goldsmith was an important contribution to our modelling techniques, since without it, even the simplest models would have proved to be infeasible to run.

Complete network

Having constructed the individual agent and server processes together with the intruder process, the overall system to be analysed is composed of a number of agent and server process instantiations in parallel with the intruder process. For example, let us assume that the system comprises one initiator agent *Alice* with nonce N_1, one responder agent *Bob* with nonce N_2, and one server *Sam* with fresh session key K. For simplicity purposes, we will also assume that each agent and server can only perform one run. Ignoring the intruder process for a moment, the network would be defined as follows:

$$\begin{array}{rcl} Network & \widehat{=} & Initiator(Alice, Sam, N_1) \\ & & ||| \\ & & Responder(Bob, Sam, N_2) \\ & & ||| \\ & & Server(Sam, K) \end{array}$$

Here, the *Responder* and *Server* processes are defined in a similar way as the *Initiator* process (on page 31). Note that these trustworthy agents and servers are interleaved, since they do not communicate with each other directly; all communication goes through the intruder. Finally, this *Network* process is then placed in parallel with the intruder process, synchronising on the *send* and *receive* events:

$$System \;\widehat{=}\; Network \underset{\{|send,receive|\}}{\parallel} Intruder'$$

where *Intruder'* is the intruder process described previously with appropriate renaming of his events to corresponding *send* and *receive* events. Hence, *System* is the actual (instantiated) process that we are interested in analysing (in this example). One must specify the number of instances of each agent or server process running the protocol and also the number of runs each of them can perform. Typically, the system being analysed will involve as many agents as are necessary for a complete run of the protocol; in the case of our Yahalom protocol, there would be two for the agents roles (for example, *Alice* as Initiator and *Bob* as Responder) and one server (for example, *Sam*). As discussed in more detail in [RSG$^+$01], Roscoe observed that no benefit is gained, from the intruder's point of view, by adding more instances of these agent roles with other identities (for example, adding another Initiator agent with identity *Carol*). The reason argued is that the intruder is designed in such a way that he is able to act like any of the agents (and can be given any number of identities to play with) and therefore can be used instead. The fact that they are not modelled as actual processes, does not limit the possible behaviour. Furthermore, as we will see in more detail in Chapter 3, Roscoe [RB99] showed that, in many cases, it suffices to provide the intruder with a single identity to play with.

One is still left with having to specify the number of runs each agent can perform and the degree of parallelism. Typically, these figures must be kept quite small to keep the analysis of the model using FDR within a feasible range. This is one of the core problems we address in this thesis.

2.2.2 Specification and verification of protocol goals

Having set up a model of the protocol like this, we have to decide what constitutes an attack, or equivalently, what specification the system hopes to satisfy. A tremendous amount of literature has been devoted to this point, for example [DvOW92, Low96b, Ros96]. This section provides an informal overview of the security properties we will be focusing on, how they are captured through appropriate specifications within our CSP models and how they are verified using FDR.

Security protocols are expected to be able to provide certain security properties in the presence of malicious intruders whose intention is to corrupt the network in such away that the security expected among the trustworthy agents running the protocol is broken; these properties will vary from one protocol to another. There are a number of security properties one may wish to obtain by using these protocols; we will be concentrating on *Secrecy* and *Authentication*.

Definition 2.1 (Secrecy) *If the data items $x_1, \ldots, x_n$ are introduced by some trustworthy agent A and intended to be shared only with agents $B_1, \ldots, B_m$ throughout some run R of the protocol, then a secrecy specification would require that these values remain secret and are never known by any participant P throughout the duration of R, where $P \notin \{A, B_1, \ldots, B_m\}$.*

■

Definition 2.2 (Authentication) *Authentication is concerned with the ability of agents to ascertain (and be assured of) who they are running the protocol with; authentication is broken if the intruder is able to convince an agent A that he has completed a run of the protocol with another agent B, when in fact it was the intruder pretending to be B. We say that A is authenticated to B to mean that if B thinks he has completed a run of the protocol with A, then A was actually running the protocol with B. There are varying levels of authentication, ranging from the bare minimum to very strict requirements (for example, the weak and injective authentication specifications defined in Section 2.3).*

■

In Section 2.3, we will give more formal definitions of these specifications and see how there are varying degrees of security in both cases.

Within our CSP protocol models, we capture these properties by adding special *signal* events to the agent processes, whose purpose is to reflect their state of mind at certain points during the protocol runs. In the older versions of our model, these signal events were explicitly included to the agent processes; we will see how this representation was optimised in Chapter 4 by removing them from this level and instead, constructing them through appropriate renaming at the system level. Using these signal events, we are then able to construct CSP processes representing the specifications we require.

To capture secrecy properties, we use the following type of signal event:

- *Claim_Secret.A.B.x* event represents the fact that the value x introduced by agent A is intended to be shared only with agent B.

This signal event would be linked to agent A's process at the point where A would expect this property to be satisfied (for example, at the end of a run). The secrecy specification then demands that if such a signal event is performed, then the secret value in question must remain unknown within the intruder's knowledge for the duration of the protocol run. It is straightforward to verify whether the intruder knows this secret or not, by extending the intruder model to include a channel *leak*, upon which he flags knowledge that should have remained secret.

To capture authentication properties, we use the following two types of signal events:

- *Signal.Running.A.B.ms* represents the fact that A believes she is running the protocol with B, agreeing upon the values in ms;
- *Signal.Commit.B.A.ms* represents the fact that B believes he has committed to a complete run of the protocol with A, agreeing upon the values in ms.

Using the Yahalom protocol as an example, let's assume we are interested in verifying whether the agent *Alice* is authenticated to *Bob*, taking the initiator and responder roles respectively. In terms of these signal events, this specification would demand that all *Signal.Commit.Bob.Alice* events, performed by *Bob* (as responder), are preceded by their corresponding *Signal.Running.Alice.Bob* events, performed by *Alice* (as initiator). Section 2.3 presents further details concerning how these signal events are constructed for the various levels of authentication.

Once we have established the necessary signal events within our protocol model and constructed the corresponding specification processes, we can use FDR to verify whether they are satisfied by our model. As described in

Section 2.1, FDR allows us to perform refinement checks between a chosen specification and implementation. For a given specification $Spec_1$ that we are interested in verifying, we perform the refinement check:

$$Spec_1 \quad \sqsubseteq_T \quad System \setminus \mathit{diff}(Events,\ SignalEvents_1)$$

where $SignalEvents_1$ is defined to be the set of all the signal events referred to by $Spec_1$; we are hiding all the events that do not correspond to the signal events being verified.

2.3 Introducing Casper

Model checkers (in our case, FDR) have proved to be extremely effective in checking for, and finding, attacks upon security protocols. However, the process of creating the CSP protocol models is time-consuming, error-prone and requires a substantial knowledge of the CSP language. Casper is a program, developed by Lowe [Low98], that takes a more abstract description of a protocol and generates the corresponding CSP description. The CSP output file is such that it can be loaded directly into FDR, and the requested checks upon the protocol automatically tested. The style of the protocol descriptions in a Casper input script is based on that in the literature and is therefore familiar to users who are interested in modelling them. Casper has is an extremely useful tool for generating these scripts and is accessible to a wide audience of users; in turn, this allows our modelling techniques for security protocols to be accessible to non-specialist users.

This section provides an overview of the basic features of Casper, upon which the subsequent chapters are dependent. There are a number of more advanced features that we will omit here, for example timed models and algebraic equivalences; a more detailed description of its complete functionality can be found in [LB00] (Casper's manual). We will continue using the Yahalom protocol presented in Example 2.1 as our standard example to introduce the format of a typical input script and illustrate how we interpret the results generated by FDR from our analysis.

2.3.1 Casper input script

Figure 2.3 presents an example of the Casper input script used to model our Yahalom protocol presented in Example 2.1. We can observe that there

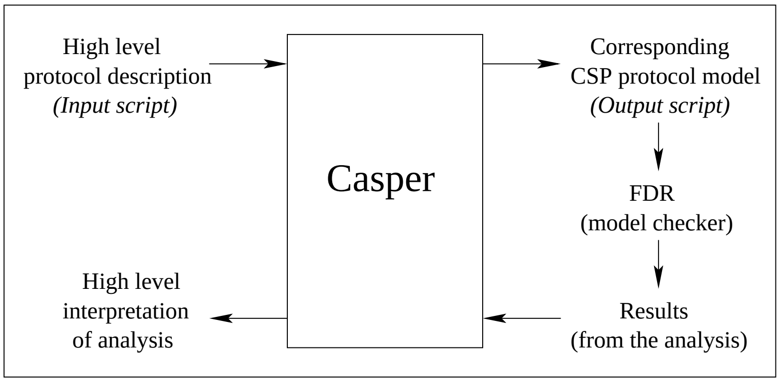

Figure 2.2: Casper - Compiler for the Analysis of Security Protocols.

are two main parts to a Casper input script. The first of these is a definition of the way in which the protocol operates in terms of the messages passed between the agents, the parameters involved (for example, agents and nonces) and their corresponding types, the specifications representing what the protocol is expected to achieve, and the initial knowledge of the agent processes declared within the model. The second part is defining the actual system to be verified against the given specifications in FDR. This includes defining the actual agent identities taking part in the system and their respective roles, instantiating all parameters within the first part of the input script, and specifying the intruder's identity together with his abilities and initial knowledge. These two main parts are divided further into eight smaller sections, each identified by the symbol #; we briefly summarise each in turn.

#Free Variables

This section declares all the types of the free variables and functions, together with the declaration of all inverse keys, used in the first part of the input script. In our example, the former includes agent and server identities, nonces, (long term) keys shared by the server and each agent (declared as a function over the agent identities) and session keys; all the keys used here are symmetric and therefore are declared as self inverse in the `InverseKeys` list.

```
-- Casper input script for the Yahalom protocol.

#Free Variables
```

Declaration of all the variables and their corresponding types, used in defining the first part of the Casper script (in this case, agent processes, protocol description and specifications) together with inverse keys.

```
A, B : Agent
S : Server
na, nb : Nonce
kab : SessionKey
SKey : Agent -> ServerKey
InverseKeys = (SKey, SKey), (kab, kab)

#Processes
```

Declaration of all the agent processes taking part. The first parameter of each process represents their identity.

```
INITIATOR(A, na) knows SKey(A)
RESPONDER(B, S, nb) knows SKey(B)
SERVER(S, kab) knows SKey

#Protocol description

0.    -> A : B
1.  A -> B : na
2.  B -> S : nb, {A, na}{SKey(B)}
3.  S -> A : nb, {B, kab, na}{SKey(A)}
4.  S -> B : nb, {A, kab, nb}{SKey(B)}
5.  A -> B : {nb}{kab}

# Specification
```

Requirements of the protocol to be verified.

```
Agreement(B, A, [])

#Actual Variables
```

Declaration of all the variables and their corresponding types, used in defining the actual system to be verified.

```
Alice, Bob, Mallory : Agent
Sam : Server
Na, Nb, Nm : Nonce
Kab : SessionKey
InverseKeys = (Kab, Kab)

#Functions

symbolic SKey
```

"symbolic" means that Casper will produce its own values to represent the results of the function applications.

```
#System
```

Definition of the actual system to be verified. In this case, our system is composed of one instance of each agent process running in parallel with each other (where each process can only perform one run).

```
INITIATOR(Alice, Na)
RESPONDER(Alice, Sam, Nb)
SERVER(Sam, Kab)

#Intruder Information

Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Sam, Mallory, Nm, SKey(Mallory)}
```

Intruder's identity within the system.

Intruder's initial knowledge.

Figure 2.3: Casper script for the Yahalom protocol defined in Example 2.1.

#Processes

This section declares all the participating roles of the protocol in question. In our example, we have an initiator, a responder and a server role. Each role description takes the form

$\texttt{Rolename(id,x}_1\texttt{,...,x}_n\texttt{) knows y}_1\texttt{,...,y}_m$

`Rolename` is the name given to the resulting CSP process representing this role. The tuple $\texttt{(id,x}_1\texttt{,...,x}_n\texttt{)}$ are the parameters of this process that must be instantiated by actual values when called upon in the system definition (in `#System`) to be analysed. The variable `id` represents the agent's identity bound to this role, whereas the other variables $\texttt{x}_1\texttt{,...,x}_n$ represent initial information required to perform a complete run of the protocol; typically, this consists of values that will be instantiated as *fresh* for each run, for example nonces and session keys, together with any identities of *mediating* roles such as servers.

The variables $\texttt{y}_1\texttt{,...,y}_m$ also represent initial knowledge required to perform a complete run; however, unlike the variables declared within the parentheses above, they are not *freshly* instantiated when agents are declared with this role in the actual system definition; they represent information that remains unchanged for each run of a given agent. A typical example of their use are long term keys (for example, keys shared with a trusted server, public keys and secret keys). They are often defined in terms of the role's identity `id`, as illustrated in our example script by `SKey(id)`.

In the Yahalom protocol example, we have declared the initiator role as follows:

```
INITIATOR(A, S, na) knows SKey(A)
```

Using this as an example, `A` is the agent identity of this role; the variable `na` is the nonce `A` will use in a protocol run, `S` is the identity of the trusted server `A` will use and `SKey(A)` is the symmetric key known solely by `A` and the server `S` (established between them before running this protocol). We will illustrate later on how these roles are instantiated in the `#System` section.

#Protocol description

This section defines the protocol itself in terms of a sequence of messages together with the corresponding sender and receiver identities for each message. The complete set of messages is regarded as a full run of the protocol. A message M may be preceded by one or more assignments, written between angle brackets. These are used in cases where the sender wishes to assign values to variables in M. A message M may also be followed by tests, written in square brackets. These consist of boolean expressions that are executed by the receiver R of M; if the test evaluates to *False*, then R aborts his run. Messages can be expressed using the following syntax:

`M, M'` represents a standard sequence of messages.

`{M}{k}` represents the encryption of message `M` by key `k` (commonly written as $\{M\}_k$).

`h(M)` represents the application of a function `h` over `M`.

`M (+) M'` represents Vernam encryption of `M` and `M'`.

`M % v` represents the case where the sender of this message sends `M`, while the receiver is prepared to accept a message of any form, storing it as the variable v.

`v % M` represents the case where the sender of this message is simply forwarding the contents bound to v, while the receiver expects a message of the form corresponding to `M`.

These last two syntax rules form an important part of Casper's expressiveness, since they enable the user to capture the notion of an agent A *forwarding* a message M, where A places no restrictions upon the contents of M; A might not be able to decrypt M and certainly has no expectation of what M should be. To illustrate this, consider the Yahalom protocol example we have been using with the following modification in messages 2 and 3 (reflecting the original form presented in [BAN89]):

Message 3. $S \rightarrow A\ :\ N_b,\ \{B, K_{ab}, N_a\}_{SKey(A)},\ \{A, K_{ab}, N_b\}_{SKey(B)}$
Message 4. $A \rightarrow B\ :\ \{A, K_{ab}, N_b\}_{SKey(B)}$

The messages in the form presented above are not a true reflection of what the protocol actually specifies; through pattern matching and the very nature of the way we assign values to the variables, A indirectly places restrictions upon the message that is encrypted under B's key, since A will expect the message to take a specific form. Furthermore, she will expect the keys assigned to the two instances of K_{ab} to be the same and similarly for the values assigned to the other variables. This is unreasonable given that A is not able (or expected) to decrypt this part of the message; A is supposed to simply forward to on to B. To avoid this, we write

Message 3. $S \rightarrow A \ : \ N_b, \{B, K_{ab}, N_a\}_{SKey(A)}, \{A, K_{ab}, N_b\}_{SKey(B)} \ \% \ v$
Message 4. $A \rightarrow B \ : \ v \ \% \ \{A, K_{ab}, N_b\}_{SKey(B)}$

This more accurately reflects the protocol's intention, since A is now prepared to accept a third component of a message 3 containing any values, store it as v and simply forward it on to B in message 4.

A more detailed description of the syntax for messages in this section can be found in [Low98].

#Specifications

This section defines the specifications that represent the security requirements for which the system will be analysed. For each specification, Casper will generate the necessary signal events within the protocol model and the corresponding CSP specification process. In turn, the resulting refinement test will be loaded into FDR and automatically verified.

The specifications we will be referring to in this thesis are those defined by Lowe [Low98] (where more details can be found concerning these and others omitted here) as follows:

`Secret(A,x,[`B_1`,...,`B_n`])` specifies that in any completed run, `A` can expect the value bound to `x` to only ever be shared with the agents in the roles $B_1, \ldots, B_n$.

`StrongSecret(A,x,[`B_1`,...,`B_n`])` is similar to the `Secret` specification above, except that it also considers incomplete runs of `A`. Thus, even if `A` has not been able to complete a run and the intruder learns the value bound to `x`, this specification will fail.

`WeakAgreement(A,B)` specifies that if `B` thinks he has successfully completed a run of the protocol with `A`, then `A` has previously been running the protocol, apparently with `B`. Note that `A` and `B` may disagree as to which role each was taking.

`Aliveness(A,B)` specifies that if `B` thinks he has successfully completed a run of the protocol with `A`, then `A` has previously been running the protocol. Note that `A` may have thought that he was running the protocol with someone other than `B`.

`NonInjectiveAgreement(A,B,[`v_1`,...,`v_n`])` specifies that if `B` thinks he has successfully completed a run of the protocol with `A`, then `A` has previously been running the protocol, apparently with `B`, and the following conditions hold true:

1. Both agents agree as to which roles they took.
2. Both agents agree as to the values bound to $v_1, \ldots, v_n$.

Note that several runs of `B` may correspond to the same run of `A`.

`Agreement(A,B,[`v_1`,...,`v_n`])` specifies that if `B` thinks he has successfully completed a run of the protocol with `A`, then `A` has previously been running the protocol, apparently with `B`, where the following conditions hold true:

1. Both agents agreed as to which roles they took.
2. Both agents agreed as to the values bound to $v_1, \ldots, v_n$.
3. There is a one-to-one relationship between the runs of `B` and the runs of `A`.

#Actual Variables

This section declares the actual variables and their types used in the second part of the input script that defines the actual system to be analysed. In our example, this includes the agent identities (`Alice`, `Bob` and `Sam`), fresh nonces (`Na`, `Nb`, `Nm`) and fresh session key (`Kab`). Once again, we must declare the inverses of all keys declared in this section; in this case, this only consists of `Kab`.

#Functions

This section consists of specifying the types of the functions used throughout the input script. A function can either be explicitly defined or symbolic (the latter being the most commonly used of the two). Explicitly defined functions are those which are defined for each argument of the function and where all values appearing on the right hand sides of the definitions are declared in the `#Actual Variables` section. Symbolic functions are those for which Casper generates its own values to represent the results of the function applications; the resulting values on the right hand side are not explicitly declared, as illustrated in our example for the function `SKey`.

#System

This section defines the actual system to be modelled and verified against the specifications in FDR. This involves instantiating the processes declared in the `#Processes` section with actual values (which have been declared in `#Actual Variables`). These processes may have multiple instances running in parallel with each other, or each process instance may be able to perform multiple sequential runs. In our example, the system is composed of one instance of the initiator role with identity `Alice`, one instance of the responder role with identity `Bob` and one instance of the server role with identity `Sam`. These processes can each perform one run using the values `Na`, `Nb` and `Kab` as their fresh nonces and session key respectively.

#Intruder Information

This section defines the intruder's identity together with the set of his initial knowledge.

2.3.2 From Casper input scripts to interpreting the results

When Casper compiles an input script, it checks the script for errors. These errors can range from typos to illogical message constructs and specification statements. Once the compilation is complete, Casper outputs the corresponding CSP script that can be directly loaded into FDR. All the refinement checks are automatically loaded and the user can request FDR to verify them. If a refinement check succeeds, then the specification in question is satisfied by the system being verified. On the other hand, if it fails, FDR will provide the user with a counter-example representing the shortest trace causing the refinement check to fail; in turn, this represents an attack upon the security specification in question. For example, the following trace is the counter-example generated by FDR and represents the authentication attack upon our Yahalom protocol example by Syverson [Syv94][2]:

```
env.Alice.(Env0,Bob)
send.Alice.Bob.(Msg1,Na)
receive.Bob.Alice.(Msg1,Na)
send.Alice.Sam.(Msg2,Sq.<Nb,Encrypt.(SKey.Alice,<Bob,Na>)>)
receive.Alice.Sam.(Msg2,Sq.<Na,Encrypt.(SKey.Alice,<Bob,Na>)>)
send.Sam.Bob.(Msg3,Sq.<Na,Encrypt.(SKey.Bob,<Alice,Kab,Na>)>)
send.Sam.Alice.(Msg4,Sq.<Na,Encrypt.(SKey.Alice,<Bob,Kab,Na>)>)
receive.Sam.Alice.(Msg3,Sq.<Nb,Encrypt.(SKey.Alice,<Bob,Kab,Na>)>)
send.Alice.Bob.(Msg5,Encrypt.(Kab,<Nb>))
```

Unless the user is familiar with the CSP protocol models and notation, this representation is not very accessible. For this reason, the counter-examples generated by FDR can be translated to a higher level description using the `interpret` (as plain text) and `linterpret` (as LaTeX source) functions in Casper. The trace above is translated to the following presentation:

$$
\begin{array}{lrcl}
\text{Message 0.} & \rightarrow \mathit{Alice} & : & \mathit{Bob} \\
\text{Message 1.} & \mathit{Alice} \rightarrow I_{\mathit{Bob}} & : & \mathit{Na} \\
\text{Message 1.} & I_{\mathit{Bob}} \rightarrow \mathit{Alice} & : & \mathit{Na} \\
\text{Message 2.} & \mathit{Alice} \rightarrow I_{\mathit{Sam}} & : & \mathit{Nb},\ \{\mathit{Bob},\ \mathit{Na}\}_{\mathit{SKey}(\mathit{Alice})} \\
\text{Message 2.} & I_{\mathit{Alice}} \rightarrow \mathit{Sam} & : & \mathit{Na},\ \{\mathit{Bob},\ \mathit{Na}\}_{\mathit{SKey}(\mathit{Alice})} \\
\text{Message 3.} & \mathit{Sam} \rightarrow I_{\mathit{Bob}} & : & \mathit{Na},\ \{\mathit{Alice},\ \mathit{Kab},\ \mathit{Na}\}_{\mathit{SKey}(\mathit{Bob})} \\
\text{Message 4.} & \mathit{Sam} \rightarrow I_{\mathit{Alice}} & : & \mathit{Na},\ \{\mathit{Bob},\ \mathit{Kab},\ \mathit{Na}\}_{\mathit{SKey}(\mathit{Alice})} \\
\text{Message 3.} & I_{\mathit{Sam}} \rightarrow \mathit{Alice} & : & \mathit{Nb},\ \{\mathit{Bob},\ \mathit{Kab},\ \mathit{Na}\}_{\mathit{SKey}(\mathit{Alice})} \\
\text{Message 5.} & \mathit{Alice} \rightarrow I_{\mathit{Bob}} & : & \{\mathit{Nb}\}_{\mathit{Kab}}
\end{array}
$$

Hence, by sending message 5, *Alice* believes she has completed the protocol run as *Initiator* with *Bob*; in reality, *Bob* never took part.

[2]This trace is slightly simplified for the purposes of illustration.

Chapter 3

Data independence techniques

In this chapter, we present an overview of the techniques we use throughout this thesis. These techniques were developed by Roscoe in [Ros98a] with the aim of enabling us to prove far more complete results regarding security protocols on model checkers than previously possible.

Over the years, model checkers have proved to be extremely effective in finding attacks upon security protocols: see, for example, [Low96a, LR97b, MCJ97, MMS97]. However, their use in *proving* protocols had generally been limited to showing that a given small instance, usually restricted by the finiteness of some set of resources such as keys and nonces, is free of attacks. Furthermore, due to the explosive effects an increase in these sets have upon the state space, they were usually restricted to very small (single) figures.

In [Ros98a], Roscoe showed how techniques borrowed from data independence and related fields can be used to achieve the illusion that nodes can call upon an infinite supply of fresh values (for example, nonces and keys), even though the actual types used for them remain finite. It is thus possible to create models of protocols in which nodes do not have to stop after a small number of sequential runs, and to claim that a finite-state run on a model checker has proved that a given protocol is free from attacks that could be constructed in the model used. One proviso is that these techniques will only detect attacks that can be formed from the given degree of parallelism; there remains the possibility that attacks could exist with a higher degree of parallelism.

This work provides one of the main foundations upon which this thesis depends. Therefore, we present a detailed overview of these techniques and the theorems we rely on here; a more complete description of this work (and details of the proofs) can be found in the original paper [Ros98a] and an extended journal version [RB99].

3.1 Introducing data independence

A process P is said to be *data independent* of the type T if it places no constraint on what T is: the latter can be regarded as a parameter of P. Typically, P can input and output members of T, store them as parameters, compare them for equality, and apply polymorphic operations to them such as tupling or listing them. Basically, P can perform any manipulations that do not require or extract any information about or is dependent on the type of T. P cannot, for example, apply comparison operators such as $<$ or $+/-$ to members of T.

The main objective of data independence analysis is usually to discover a finite *threshold* for a verification problem parameterised by the type T: a size of T beyond which the answer (positive or negative) to the problem will not vary. If we can verify a system for the threshold size of T, then we will have verified it for all larger values of T. This can be done successfully for a wide range of problems, as is shown, for example, in [Ros98b, Laz99].

There are a number of features displayed by security protocol models which could be regarded and thus treated as data independent entities. The main features are the sets of keys and nonces which are represented as parameters of the processes underlying the model. Depending on the protocol, there could be additional sets with similar properties as these, for example index values in the Otway-Rees protocol. These data types are not bound to a specific user and they are treated in such a way that the users do not need to know or are not dependent on the specific types of the values. In addition, the abstract constructs used in the models are polymorphic. For example, a typical construct of a message 2 in the Yahalom protocol (using the version presented in Example 2.1, Chapter 2) is captured as:

```
Sq.<Nb, Encrypt.(SKey(B), Sq.<A, Na>)>
```

Clearly this construct does not depend upon the type of `Nb`, `Na`, `A` or `B` and does not reveal additional information concerning these types.

Unfortunately, the direct results of data independence based on computing thresholds cannot be directly applied to the security protocol models.

The problem lies with the characteristic that security protocols rely on the uniqueness of the nonces and keys (and any other similar data objects depending on the protocol). The model must remember all values in use and previously used in order to generate distinct data values. To ensure that the intruder is not unreasonably restricted, the model must remember all old data values and messages sent. This allows him to perform any logical deductions based on all possible knowledge that he could have gained. To fulfill these requirements, the models must consist of an unbounded storage of values. Since the calculation of thresholds in the case of processes such as security protocols (consisting of equality tests) is primarily based on finding the maximum number of values a process will ever need to store, *finite* thresholds cannot be computed for the data objects defined in these models.

3.2 Collapsing functions

Since the general-purpose data independence results of the earlier papers had proved to be inapplicable, the approach (presented in [Ros98a]) was to apply some of the methods underlying the proofs of these results (derived from work done by Lazić and Roscoe [LR97a]) directly to the sort of CSP model that a protocol analysis generates. The aim was to take a "full-sized" model of a protocol (one with an unbounded number of other agents, and infinite sets of nonces, etc.) and to use these methods to reduce the problem of proving the correctness of a corresponding system for all parameter values to a finite check.

3.2.1 Underlying Principles

The underlying principle upon which the resulting methods are based involves applying a collapsing function ϕ to the parameter type T of a process P, and proving that the behaviour of $P(T)$ before the mapping, is *lifted* through the function and is therefore a subset of the behaviour of $P(\phi(T))$. By behaviour we are specifically interested in the traces of the processes modelled. It is intuitive to see that if the collapsing function ϕ is injective and is applied to parameter type T of process P (where P is data independent of T), then the behaviour *before* the mapping is equivalent to the behaviour *after* the mapping:

$$traces\,(P(\phi(T))) \quad = \quad \{\phi(t) \mid t \in traces\,(P(T))\} \tag{3.1}$$

since all members of T are mapped to distinct values. A problem arises when we have a non-injective function, since it can change the results of whatever equality tests occur within the program. In other words, it might map an execution in which two distinct values are input and then compared for equality, into one where two equal values are input and compared for equality. This is illustrated in the following example.

Example 3.1 (Variables) *Consider the following process:*

$$\begin{aligned} P(T) \;\widehat{=}\; & in?x : T \rightarrow in?y : T \rightarrow \mathit{if}\ (x = y)\ \mathit{then}\ a \rightarrow STOP \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ b \rightarrow STOP \end{aligned}$$

Consider the instance of this where $T = \{0, 1\}$ and ϕ is a non-injective function over T defined as:

$$\phi(0) = \phi(1) = 0$$

In this example, we have the following sets of traces:

$$\begin{aligned} \mathit{traces}\,(P(\phi(T))) &= \{\langle\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\} \\ \mathit{traces}\,(P(T)) &= \{\langle\rangle, \langle in.1\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.1\rangle, \\ &\quad \langle in.1, in.0\rangle, \langle in.1, in.1\rangle, \langle in.0, in.0, a\rangle, \\ &\quad \langle in.1, in.1, a\rangle, \langle in.0, in.1, b\rangle, \langle in.1, in.0, b\rangle\} \\ \{\phi(t) \mid t \in \mathit{traces}\,(P(T))\} &= \{\langle\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle, \\ &\quad \langle in.0, in.0, b\rangle\} \end{aligned}$$

Clearly,

$$\mathit{traces}\,(P(\phi(T))) \neq \{\phi(t) \mid t \in \mathit{traces}\,(P(T))\}$$

■

There may be no relationship at all between the subsequent behaviours in these two cases, and the equivalence in (3.1) becomes the possibly strict inequality

$$\mathit{traces}\,(P(\phi(T))) \subseteq \{\phi(t) \mid t \in \mathit{traces}\,(P(T))\} \qquad (3.2)$$

This inequality applies to variables only; in the case of constants in the program, this is not necessarily the case, as illustrated in the following example.

Example 3.2 (Constants) *Consider the following process:*

$$
\begin{array}{l}
P(T, c_1, c_2) \ \widehat{=} \\
\qquad in?x \ \rightarrow \ if \ (x = c_1 \ and \ x = c_2) \ \ then \ left \, . \, x \ \rightarrow \ STOP \\
\qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \quad else \ \ right \, . \, x \ \rightarrow \ STOP
\end{array}
$$

where c_1 and c_2 are both constants over the type T of $P(T, c_1, c_2)$. Let us assume that $T = \{1, 2\}$, c_1 and c_2 are assigned the values 1 and 2 respectively and ϕ is a non-injective function over T defined as:

$$\phi(1) \ = \ \phi(2) \ = \ 0$$

For this example, we have the following sets of traces:

$$
\begin{array}{rcl}
traces\,(P(\phi(T), \phi(1), \phi(2))) & = & \{\langle\rangle, \langle in.0\rangle, \langle in.0, left.0\rangle\} \\
traces\,(P(T, 1, 2)) & = & \{\langle\rangle, \langle in.1\rangle, \langle in.2\rangle, \langle in.1, right.1\rangle, \\
 & & \langle in.2, right.2\rangle\} \\
\{\phi(t) \mid t \in traces\,(P(T, 1, 2))\} & = & \{\langle\rangle, \langle in.0\rangle, \langle in.0, right.0\rangle\}
\end{array}
$$

The inequality (3.2) *is not satisfied since*

$$\langle in.0, left.0\rangle \quad \not\in \quad \{\langle\rangle, \langle in.0\rangle, \langle in.0, right.0\rangle\}$$

■

The PosConjEqT condition

PosConjEqT (Positive Conjunctions of Equality Tests) is a condition, derived by Lazić [Laz99], that regains the equality in (3.1), even when non-injective functions are applied to variables in a given program.

Definition 3.1 (PosConjEqT) *A program P satisfies PosConjEqT precisely when the failure of each equality check E in P results in the process STOP; the result of an equality test proving false will never result in the program performing a trace that it could not have performed were the test to prove true.*

■

The following example illustrates the effect of this condition (upon variables).

Example 3.3 (PostConjEqT over variables) *Consider the following process:*

$$P(T) \;\widehat{=}\; in?x : T \;\rightarrow\; in?y : T \;\rightarrow\; \mathit{if}\ (x = y)\ \mathit{then}\ a \rightarrow STOP \\ \mathit{else}\ \ STOP$$

Let us assume that $T = \{0, 1\}$ *and* ϕ *is a non-injective function over* T *defined as:*

$$\phi(0) \;=\; \phi(1) \;=\; 0$$

For this example, we have the following sets of traces:

$$\begin{aligned}
\mathit{traces}(P(\phi(T))) &= \{\langle\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\} \\
\mathit{traces}(P(T)) &= \{\langle\rangle, \langle in.0\rangle, \langle in.1\rangle, \langle in.0, in.0\rangle, \langle in.0, in.1\rangle, \\
&\quad\ \langle in.1, in.0\rangle, \langle in.1, in.1\rangle, \langle in.0, in.0, a\rangle, \\
&\quad\ \langle in.1, in.1, a\rangle\} \\
\{\phi(t) \mid t \in \mathit{traces}(P(T))\} &= \{\langle\rangle, \langle in.0\rangle, \langle in.0, in.0\rangle, \langle in.0, in.0, a\rangle\}
\end{aligned}$$

The equality in (3.1) *is regained.*

■

In the case of constants in a program P, where P satisfies $PosConjEqT$, the equality (3.1) is not regained; instead, the following inequality is guaranteed:

$$\{\phi(t) \mid t \in \mathit{traces}\,(P(T))\} \;\subseteq\; \mathit{traces}\,(P(\phi(T))) \tag{3.3}$$

Example 3.4 (PostConjEqT over constants) *Consider the following process:*

$$
\begin{aligned}
& P(T, c_1, c_2) \ \widehat{=} \\
& \qquad in?x \ \rightarrow \ if \ (x = c_1 \ and \ x = c_2) \ then \ left.x \ \rightarrow \ STOP \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ else \ \ STOP
\end{aligned}
$$

where c_1 *and* c_2 *are both constants over the type* T *of* $P(T, c_1, c_2)$*. Let us assume that* $T = \{1, 2\}$*,* c_1 *and* c_2 *are assigned the values* 1 *and* 2 *respectively and* ϕ *is a non-injective function over* T *defined as:*

$$\phi(1) \ = \ \phi(2) \ = \ 0$$

For this example we have the following sets of traces:

$$
\begin{aligned}
traces\,(P(\phi(T), \phi(c_1), \phi(c_2))) \ &= \ \{\langle\rangle, \langle in.0\rangle, \langle in.0, left.0\rangle\} \\
traces\,(P(T)) \ &= \ \{\langle\rangle, \langle in.1\rangle, \langle in.2\rangle\} \\
\{\phi(t) \mid t \in traces(P(T))\} \ &= \ \{\langle\rangle, \langle in.0\rangle\}
\end{aligned}
$$

The inequality (3.3) *is satisfied, whereas the equality* (3.1) *is not.*

■

The PosConjEqT$'_C$ condition

Some aspects of the protocol examples, such as the ways in which agent processes handle keys, nonces and identities, fit into this *PosConjEqT* framework. One exception to this is that agent processes may well perform inequality checks with a few constants, such as their own names. Additionally, the presence of a parallel process whose alphabet and communications involve a given constant like this will generally introduce such inequality checks implicitly, as will the special-case of constants belonging to the intruder's initial knowledge. For example, the composition

$$P \underset{\{|send.Bob|\}}{\parallel} Q$$

where *Bob* is a constant of the data independent type Agent, requires the equality test $x = Bob$ to determine whether the event $send.x$ is synchronised upon or not. Finally, a process involving constants can legitimately depend on their mutual inequality even when it does satisfy *PosConjEqT*, and it would then be equally inappropriate to apply a function ϕ that identifies these values. For example, the agent process *Bob* could be implemented such that he can run the protocol with any other agents, except for himself. In this case, events are enabled when the equality check of identity with himself fails. Thus, a condition $PosConjEqT'_C$ is defined for C a set of constants as follows:

Definition 3.2 (PosConjEqT'_C) *A program P satisfies $PosConjEqT'_C$ for a set of constants C, precisely when P satisfies PosConjEqT with the following exception: P may have non-*`STOP`*results for equality tests involving at least one member of the set C of constants. The collapsing result above holds provided ϕ is faithful to the values of the constants in C:*

$$\forall\, x \in T, c \in C \;\bullet\; \phi(x) = \phi(c) \;\Leftrightarrow\; x = c$$

where T is some data independent type upon which the collapsing function ϕ can be applied.

■

3.2.2 Positive deductive systems

The agent processes in a protocol model are fairly standard-style processes and easy to check for properties like $PosConjEqT'_C$. The intruder process, however, is somewhat more complex by its very nature, relying on a *deductive system* that specifies the rules concerning the making and breaking of messages. It is frequently the case that a deduction (X, f) will only be able to occur when two objects of some type T are equal, for example the standard (symmetric) decryption deduction

$$\{M\}_K,\; K \;\;\vdash\;\; M$$

carries an implicit equality check between the key in the encryption and the free-standing key. This is entirely within the spirit of *PosConjEqT* and has the crucial property that the identification of two keys by a function will never disable a deduction. However, it is equally possible to construct deductions that do not satisfy this property, as illustrated by the following example deduction:

$$\{M\}_{K_1},\ \{M\}_{K_2},\ K_1 \neq K_2 \ \ \vdash \ \ M \tag{3.4}$$

Here, the equality test between the two keys K_1 and K_2 resulting in true disables this deduction from taking place, while the failure of this equality results in positive action occurring (in the sense that the deduction is enabled).

Definition 3.3 (Positive deductive system) *A deductive system is said to be positive relative to some type parameter T if, for any function between types $\phi : T_1 \to T_2$, whenever (X, f) is an inference the system generates for type T_1, then $(\phi(X), \phi(f))$ is one generated for T_2. What this essentially requires is that the generation of the deductions is symmetric in T (i.e. treats all members of T equivalently) and never has an inequality requirement over members of T between places they appear on the left-hand side of a deduction.*

■

From this, the following property is established: if an intruder is built over a positive deductive system, and its initial knowledge set contains no inequality tests (explicit or implicit) with members of our type T other than the constants C, then the resulting process (and hence, subject to obvious conditions on the other parts of the network, the entire protocol model) satisfies $PosConjEqT'_C$.

Being able to show that components of a system satisfy these properties plays a crucial part in the strategy developed for constructing more complete proofs within our CSP framework of protocol analysis, as will become clear in the following section.

3.3 Application to CSP protocol models

The aim was to create the illusion of having an unbounded supply of fresh values (for example, nonces and session keys - essentially all values that are expected to be fresh in every run), where the actual types of these values remain finite. This would make it possible to claim that a protocol of unbounded size (nodes performing unbounded number of sequential runs) is free from attacks, by performing a finite refinement check in FDR.

This is achieved by applying transformations of the non-injective nature to a protocol of unbounded size, and reducing the problem to a corresponding finite system. Roscoe proves that if all the components of the system obeys the $PosConjEqT'_C$ condition, then the complete behaviour (in terms of the system's traces) of the unbounded system *before* the transformations is fully captured by the corresponding finite system *after* the mappings have been applied. In this section, we give an overview of this method.

Each time a node generates a fresh value of some type T, it immediately communicates it within a message to the intruder (acting as the unreliable medium). Rather than have these nodes specifying which fresh values they choose, it is possible to delegate to an external process, known as a *Manager* process; this does not influence the analysis, but can be viewed as an artifact of the modelling techniques we use. Every time a process wants to create a fresh value v with an output communication, the communication is turned into an input of v and forced to synchronise with the corresponding manager process (in addition to all the other processes it is synchronising with). To satisfy the uniqueness and freshness properties expected for such values, the manager processes would have to be implemented such that they remember all the fresh values given out and never issue the same one twice or the one belonging to the intruder at all. Clearly such an implementation would not satisfy the $PosConjEqT$ condition. As a protocol system runs, the trustworthy nodes will only ever hold a small finite number of fresh values at any one time (since, it seems reasonable to assume that as soon as they close, they forget the values regarded as fresh in the previous runs). However, the intruder and the manager processes will both be able to store an unbounded number; the manager processes remembers all the ones seen so far so that it does not repeat itself, and the intruder remembers everything it has seen (including objects that contain an arbitrary number of fresh values) so that this can potentially be exploited in future attacks.

To reduce such a system to a corresponding *finite* one (that can then be analysed using FDR), there is no reason why one cannot apply a transformation function ϕ to a data independent program in the middle of a run. As described in [Ros98a], suppose P is such a program satisfying $PosConjEqT$. Any state P' that P may have reached during its execution (i.e. P' is a state reached in its operational semantics, namely a program that represents how P behaves after some sequence of actions) will still be a data independent program, though it could well have acquired some values from T that it holds in its identifiers which were not present in its initial state. For a given assignment to these identifiers, we will always have

$$traces\,(\phi(P')) \quad \supseteq \quad \{\phi(t) \mid t \in traces\,(P')\} \tag{3.5}$$

because of the *PosConjEqT* property. The inequality may be strict because some values held in P' that are distinct, may get mapped to the same place by ϕ, resulting in an equality test in $\phi(P')$ giving the answer *true* where it gave *false* in P'. The values that are bound to variables at the point ϕ is applied, behave like the constants in Example 3.4, as illustrated by the following example.

Example 3.5 *Consider the following process:*

$$\begin{array}{l} P(T, v_1, v_2) \;\widehat{=} \\ \qquad in?x \;\rightarrow\; if\;\; (x = v_1 \;\; and \;\; x = v_2) \;\; then \;\; left.x \;\rightarrow\; STOP \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; else \;\;\; STOP \end{array}$$

where v_1 and v_2 are state variables of data independent type T and T is defined as the set $\{1, 2\}$. Now consider a state P_n that P has reached, where v_1 and v_2 are bound to the values 1 and 2 respectively. If, in state P_n, we apply the same non-injective function ϕ over T as defined in Example 3.4, namely:

$$\phi(1) \;=\; \phi(2) \;=\; 0$$

then the resulting behaviour is the same as that in Example 3.4 (using constants):

$$\begin{aligned} traces\,(P_n(\phi(T), \phi(1), \phi(2))) \;&=\; \{\langle\rangle, \langle in.0, left.0\rangle\} \\ traces\,(P_n(T, 1, 2)) \;&=\; \{\langle\rangle, \langle in.1\rangle, \langle in.2\rangle\} \\ \{\phi(t) \mid t \in traces(P_n(T, 1, 2))\} \;&=\; \{\langle\rangle, \langle in.0\rangle\} \end{aligned}$$

■

Transformations are then applied that identify all *forgotten* fresh values, no longer known to any trustworthy nodes, but remembered by the intruder. These transformations are implemented into the intruder (whose memory gets transformed each time) and the manager processes, that can now re-use the values that the intruder has *forgotten* through the transformation. As long as the number of distinct values remembered by the intruder is bounded, by applying transformations whenever it gets too large, the models are able to generate an unlimited supply of fresh values from a finite set.

These transformations do not directly affect the states of the trustworthy node processes, but can potentially increase the traces of the intruder because of the above inequality, thereby leading to potential false attacks.

Example 3.6 (False attack) *Consider the case where the intruder has overheard the encrypted message $\{M\}_{K_1}$, where M is a secret and K_1 is a symmetric key. Let us assume that the intruder does not know K_1 and is therefore unable to decrypt this secret message; he does, however, store another key K_2. Under normal modelling assumptions (unless specified otherwise), K_2 would not help him decrypt this message directly. We will assume that this holds for the purposes of our example.*

Now consider what happens if we apply the following collapsing function ϕ upon these session keys throughout the intruder's memory:

$$\phi(K_1) \;=\; \phi(K_2) \;=\; K_3$$

This would transform the message $\{M\}_{K_1}$ to $\{M\}_{K_3}$ and the key K_2 to K_3, thereby allowing the intruder to decrypt it and learn the secret message M through the simple decryption deduction:

$$\{M\}_{K_3},\; K_3 \;\vdash\; M$$

Clearly, this is a false attack as a result of the modelling techniques, rather than a reflection of a genuine attack upon the protocol itself.

■

This method is proved to be *fail-safe*, leading to the following proposition:

Proposition 3.1 *Suppose one of our protocol models has (i) a positive deductive system, (ii) the trustworthy processes satisfy $PosConjEqT$ for some data independent type W and (iii) the trace specification S does not explicitly constrain W; so the specification is judged by the check $S \sqsubseteq_T P \backslash X$, where X includes all the events (communications) involving W. Let $P^\dagger$ be a model of the protocol with the addition of events collapse.ϕ which, when executed, apply ϕ to all values of type W held in state. Further suppose that $P^\dagger$ uses a finite type W, but is always able to perform a proper (non-injective) collapse when it reaches a state where the whole of W is in use. Then,*

$$S \sqsubseteq_T P^\dagger \backslash X \quad \Rightarrow \quad S \sqsubseteq_T P \backslash X$$

where P is the same protocol model without collapses but with arbitrary (including infinite) W.

■

By fail-safe, we mean that the method will guarantee to find an attack if there is one, but may still fail even when there is no attack because a transformation has enabled the intruder to perform an inference it could not have performed otherwise. Whether or not such false attacks are thrown up is heavily affected by the strategy that is used for generating the transformations applied during execution: we discuss this in more detail in Chapter 4.

3.4 Specification considerations

As we discussed in the previous sections, when we apply the transformation techniques to a system's implementation, we potentially increase the set of traces. We know that this does not pose a problem when applying them to the implementation side of the traces refinement check. However, we cannot simply apply these transformations to the specification side of the refinement check, since the potential increase in traces would weaken the specification, leading to potential false-positives. In the work we have discussed so far, this problem was avoided by hiding all events that refer to values upon which transformations were defined on the implementation side, and not make any references to them on the specification side (as illustrated in Proposition 3.1).

Analysing protocol models under these assumptions is indeed the simplest approach to handling this problem and was used in our initial application work for the case studies presented in [RB99]. This does, however, place a large constraint on how expressive the specification can be, since we are unable to refer implicitly to any values of the data type in question. The specification style we would like to use is that used by Casper. As discussed in more detail in Chapter 2, these specifications are defined in terms of special signal events that reflect the state of mind of agents, with regards to what values they believe are bound to certain variables. These will normally involve agent identities, upon which we do not apply transformations, and variables that are expected to be fresh or unique. The latter are precisely the sorts of values upon which we would typically apply the collapsing functions. The following example illustrates the potential problem caused when we include the collapses of values on the left and right side of the refinement.

Example 3.7 *Consider the following pair of processes*

$$\begin{array}{rcl} P_1 & = & a?x : T \rightarrow a?y : T \rightarrow b\,.\,x \rightarrow STOP \\ P_2 & = & a?x : T \rightarrow a?y : T \rightarrow b\,.\,y \rightarrow STOP \end{array}$$

Clearly, P_1 does not refine P_2. For example, if T is defined as the set $\{0, 1\}$, then a counter example would be the trace $\langle a\,.\,0, a\,.\,1, b\,.\,0\rangle$. If we allowed an on-the-fly collapse of the data type after the two inputs, there is the danger that the two input values might be identified resulting in a false positive answer.

■

There are in fact two types of dangers that need to be considered. The first is that through a collapse, the traces of the specification itself might be increased (by changing the answer to an equality test). The second, and the one displayed by the example above, is that a collapse might result in equating a value held in an identifier in the specification with one in the implementation. Either could lead to a false positive in a refinement check.

These twin dangers are avoided by deriving a set of conditions upon the use of collapsing functions within specifications to ensure that they are not weakened. These were derived as follows:

Consider a specification process S that applies to a system P without collapses. The aim is to create a version $S^\dagger$ that applies to $P^\dagger$ in which each collapse that occurs is displayed in the trace, without the danger of false positives.

Unless specified otherwise, one needs to consider the case where any collapse ϕ can occur in all the states of S. Suppose $S_n(X)$ is such a state, where X represents all the parameters acquired at this point in the run. The corresponding state $S_n^{\dagger}(X)$ of $S^{\dagger}$ will have all the actions of $S_n(X)$ plus, for each collapse ϕ, the action *collapse*.ϕ leading to a state $S_m^{\dagger}(X')$ that satisfies the following properties:

(a) $traces\, S_m^{\dagger}(X') \;\subseteq\; \phi(traces\, S_n^{\dagger}(X))$

(b) X' contains no value in T which is the image under ϕ of more than one thing.

Based on these conditions, the following proposition is established:

Proposition 3.2 *Suppose that P and $P^{\dagger}$ are defined according to Proposition 3.1, and S, $S^{\dagger}$ are defined as above. Then*

$$S^{\dagger} \sqsubseteq_T P^{\dagger} \;\;\Rightarrow\;\; S \sqsubseteq_T P$$

■

Informally, at each stage either $P^{\dagger}$ can perform an analogous action or a collapse followed by an analogous action, and the effect of the collapse is to make $S^{\dagger}$ (if anything) more stringent and $P^{\dagger}$ (if anything) worse. Condition (b) above ensures that any equalities existing *after* the collapse between values in the states of $S^{\dagger}$ and $P^{\dagger}$, also held *before*.

The latter condition may, in many applications, be an uncomfortably strong condition, but fortunately it matches the way we model protocols well in most cases. We discuss how the specifications are extended within the CSP protocol models to incorporate the data independence techniques (satisfying these conditions above) in Chapter 4.

3.5 Summary

This chapter presented an overview of the data independence techniques developed by Roscoe [Ros98a], that have enabled us to prove far more complete results using model checkers than hitherto. These techniques were derived from the data independence work by Lazić in [LR97a, Laz99], and consist of applying collapsing functions throughout the run of a protocol under a series of special conditions. These conditions proved to be a natural property of our CSP protocol models and have the effect of being able to prove that any attack existing upon the protocol of unbounded size will also be captured in the corresponding reduced (and finite) system after the transformations have been applied. By unbounded protocol, we mean that the participating agents are able to perform an unbounded number of sequential runs each (thereby requiring an unbounded supply of fresh values). The illusion of having an unbounded supply of fresh values, when in fact the actual type of these values remain finite, is achieved by applying collapsing functions upon these fresh values (mapping them to representative stale values) within the intruder's memory at certain points throughout the run of the system (once they have been forgotten by the agent processes). This has the effect of *recycling* these values and allowing them to be supplied as fresh over and over again.

A further result that Roscoe achieved using these techniques was to formally prove that, under a few natural conditions, it suffices to give the intruder a single identity. This is a useful result, since almost all protocol models for model checkers that one sees make this simplification.

While these techniques enable us to prove the correctness of protocols where the agents can perform an unbounded number of sequential runs each, it does not address the issue of unbounded number of instances of agents in parallel; if no attack is found within our models, it does not prove than none exist for a higher degree of parallelism within each agent identity. We address this issue in Chapter 6.

These techniques have proved to be applicable to a range of protocols, as illustrated in [RB99] and demonstrated throughout this thesis. A class of protocols that these data independence techniques are not able to directly handle (under the assumptions presented in this chapter), are those involving timestamps. Types such as timestamps present a problem because the typical operations performed is out of the range of data independence; for example, using comparison operators such as $x < y$ to determine whether the timestamp y is *older* than x. Extending this work to handle timestamps is not an area that we address in this thesis. However, we consider this an interesting avenue for future research.

Chapter 4

Automating data independence

Chapter 2 gave an overview of the CSP framework we use to model and analyse security protocols. This method, and model checkers in general, have proved to be extremely effective in finding attacks on security protocols. However, their use in *proving* protocols have generally been limited to showing that a given small instance, usually restricted by the finiteness of some set of resources such as keys and nonces, is free of attacks.

The data independence techniques presented in Chapter 3 has made it possible to prove far more complete results on model checkers than hitherto, since it allows us to simulate a system where agents can call upon an infinite supply of different nonces, keys, etc., even though the actual types remain finite. It is thus possible to create models of protocols in which agents can perform an unbounded number of sequential runs (although with a fixed, finite number of *concurrent* runs) and therefore claim that a finite-state check on a model checker, such as FDR [For97], proves that a given protocol is free from attacks, subject to the same limit on the number of concurrent runs. Later on in Chapter 6, we will address this restriction of only being able to model a finite number of concurrent runs and present a possible solution for making our methods even more complete.

In this chapter, we start by presenting the design and implementation of these data independence techniques within our CSP protocol models (Section 4.1). Whilst developing these techniques, it soon became apparent that these techniques cause a substantial increase in the state space required per protocol model, than was previously the case without them. For this reason, we developed a series of effective state space reduction strategies that address this problem. We discuss these strategies and the degree of improvement they provide in Section 4.2 of this chapter.

In Section 4.3, we present the automation of the data independence techniques by integrating them into Casper, thereby making them more accessible. Finally, we present an example in Section 4.4, where we apply these techniques to the Otway-Rees protocol [BAN89]. This is a useful illustration of (i) how these techniques provide a more thorough analysis compared to our previous models, and (ii) how effective (and essential for feasibility purposes) the state space reduction strategies are.

We use the corrected Needham-Schroeder Public-Key (NSPK) protocol (by Lowe in [Low96a]) as the main example throughout this chapter, defined as follows:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B & : \{A,\ n_a\}_{PK(B)} \\ \text{Message 2.} & B \rightarrow A & : \{n_a,\ n_b,\ B\}_{PK(A)} \\ \text{Message 3.} & A \rightarrow B & : \{n_b\}_{PK(B)} \end{array}$$

where A and B are agents introducing the fresh nonces n_a and n_b respectively. The aim of this protocol is to authenticate A and B with one another, through the exchange of these nonces.

4.1 Extending CSP protocol models

In this section, we discuss the extensions made to our CSP protocol models to incorporate the data independence techniques presented in Chapter 3.

These allow us to simulate a system where agents can call upon an unbounded supply of fresh values (such as nonces and keys) even though the actual types remain finite. Roscoe achieved this by removing the responsibility of generating fresh values from the trustworthy and the intruder processes, and introducing a new special process, known as a *Manager* process. This process is responsible for supplying the system with an unbounded supply of fresh values when required. It was observed that a trustworthy agent (or server) A will typically store values that are regarded as fresh, for the duration of a single run. So the assumption made is that A *forgets* all stored fresh values at the point A withdraws or completes a run of the protocol. We say that a fresh value v is *forgotten* precisely when v is no longer known (stored) by any trustworthy participants. The only component that never forgets these values is the intruder, since he stores all new messages seen across the network. It is precisely upon these fresh values stored in the

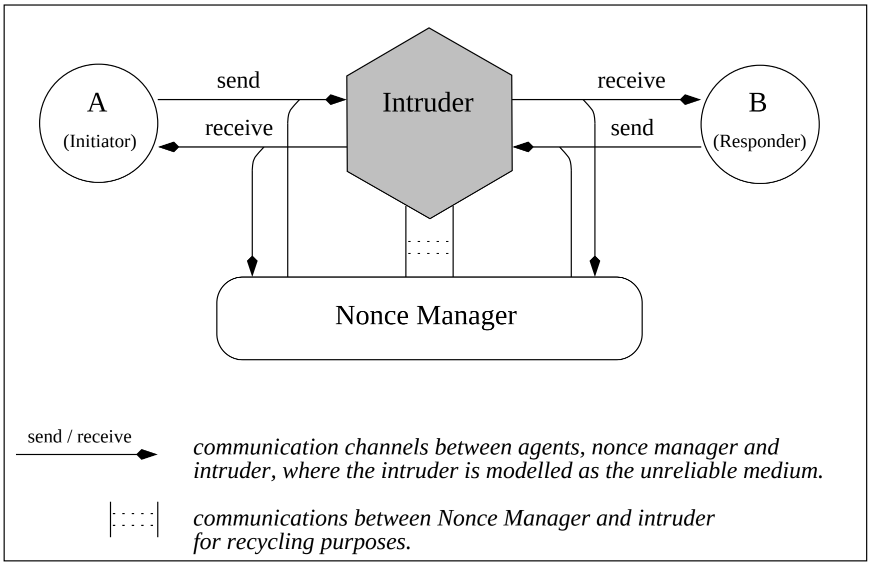

Figure 4.1: Network in CSP models that incorporate the data independence techniques, using the NSPK protocol as the example.

intruder's memory that the collapsing functions are applied (as described in Chapter 3). We refer to this mapping process as the *recycling mechanism* of fresh values. Hence a fresh value v is *recycled* precisely when v has been *forgotten* within the network. Once such a value has been recycled, the corresponding manager process can supply it to the network as fresh again. It is this recycling mechanism that enables us to create the illusion of having an infinite supply of fresh values from a finite source.

We discuss the extensions required to the individual components of our CSP protocol models in turn, together with the new addition of the manager processes. Figure 4.1 illustrates the network of processes in our extended models using the adapted NSPK protocol described above as an example.

4.1.1 Trustworthy processes

In Chapter 2, the system within the CSP protocol model to be analysed was composed of a number of instantiated trustworthy agent (or server) processes and an intruder process. For example, in the NSPK protocol specified above, if we wanted to model one instance as the *Initiator* role (A) and one as the

Responder role (B), where they could perform a single run of the protocol each, then we would have to instantiate these processes accordingly, each with an identity and a fresh nonce. In Casper this would be captured as the following system:

$$\begin{array}{l} \mathit{Initiator}(\mathit{Alice}, N_A) \\ \mathit{Responder}(\mathit{Bob}, N_B) \end{array}$$

where the values N_a and N_b represent the fresh values for *Alice* and *Bob* respectively.

In the case of our CSP models that use the data independence techniques, the aim is to remove the instantiation of variables that are expected to be fresh for each run; in our example, this refers to the nonces only. Instead, the responsibility of supplying fresh values when required will be passed on to the corresponding *Manager* processes. The agent processes are free to perform any number of sequential runs they want; it is the responsibility of these manager processes to know when they require fresh values and supply them accordingly. Thus, as we will see in more detail later on in Section 4.3, the system above would now be captured in Casper as:

$$\begin{array}{l} \mathit{Initiator}(\mathit{Alice}) \\ \mathit{Responder}(\mathit{Bob}) \end{array}$$

with the goal of allowing both agents to perform an unbounded number of sequential runs each.

Thus, the extensions required to the agent processes in the CSP models to incorporate the data independence techniques are very minor. Previously, the *Initiator* role would have been modelled as the following process:

$$\begin{array}{l} \mathit{Initiator}(A,\ n_a) \mathrel{\widehat{=}} \\ \quad \Box\, B : \mathit{Agent} \ \bullet \ \mathit{env} \,.\, A \,.\, B \ \rightarrow \\ \qquad \mathit{send} \,.\, A \,.\, B \,.\, \{A,\ n_a\}_{PK(B)} \ \rightarrow \\ \qquad \Box\, N_b : \mathit{Nonce} \ \bullet \\ \qquad\quad \mathit{receive} \,.\, B \,.\, A \,.\, \{n_a,\ n_b,\ B\}_{PK(A)} \ \rightarrow \\ \qquad\quad \mathit{send} \,.\, A \,.\, B \,.\, \{n_b\}_{PK(B)} \ \rightarrow \ \mathit{SKIP} \end{array}$$

In our new CSP protocol model, we extend this process as follows:

$$
\begin{array}{l}
\mathit{Initiator}(A) \mathrel{\widehat{=}} \\
\quad \Box\, B : \mathit{Agent} \;\bullet\; \mathit{env} \,.\, A \,.\, B \;\rightarrow \\
\qquad \Box\, n_a : \mathit{Nonce}_F \;\bullet\; \mathit{send} \,.\, A \,.\, B \,.\, \{A,\; n_a\}_{PK(B)} \;\rightarrow \\
\qquad\quad \Box\, n_b : \mathit{Nonce} \;\bullet \\
\qquad\qquad \mathit{receive} \,.\, B \,.\, A \,.\, \{n_a,\; n_b,\; B\}_{PK(A)} \;\rightarrow \\
\qquad\qquad \mathit{send} \,.\, A \,.\, B \,.\, \{n_b\}_{PK(B)} \;\rightarrow \\
\qquad\qquad \mathit{close} \,.\, A \,.\, \mathit{Initiator} \;\rightarrow\; \mathit{Initiator}(A)
\end{array}
$$

In summary, there are three main changes we have made. Firstly, the new process definition is a recursive one, reflecting the fact that the *Initiator* role can perform an unbounded number of sequential runs; in the former model, the process terminated in *SKIP*. Secondly, the fresh nonce n_a is no longer a parameter, but instead is taken from some set Nonce_F (by definition, the process is prepared to accept any value in this set); we will see later on how the supply of these values are determined by the manager processes. Thirdly, there is an additional *close* event that the process performs before recursing back to its initial state. This event flags the end of a run of this process and plays an important role in the recycling mechanism of fresh values; this will be discussed in more detail below.

4.1.2 Manager processes

As introduced briefly above, *Manager* processes are responsible for providing the network with an unbounded supply of fresh values from a finite source. We define one manager process for each data independent type; in our corrected NSPK protocol example, we would define one, referred to as the *Nonce Manager*, to handle the flow of values of type *Nonce*. In this section, we give a brief overview of the CSP design and implementation of these processes.

For each data independent type T within the protocol model, there are two sets of values declared. The first consists of *foreground* values that are regarded as fresh by the network. The corresponding manager of T uses this set to provide the network with a continuous supply of fresh values of type T. The minimum number of fresh values required to achieve this continuous flow is dependent upon the protocol being modelled; for our adapted NSPK protocol example, the nonce manager requires three foreground values. We will see further on in Section 4.3 how we ensure that our managers are given sufficient values. The second set consists of *Background* values, representing the old or *stale* values of type T. It is this latter set of values that the fresh ones are mapped to (within the intruder's memory) when they get recycled.

The manager component for some type T is modelled as a highly parallelised process in our CSP models, for efficiency purposes during the compilation phase when running them through FDR. At the top level, we define it as the parallel composition of two processes, namely, the *Monitor and Trigger Recycling* (*MTR*) and the *Supply* processes.

$$Manager_T \;\widehat{=}\; MTR_T \underset{X}{\parallel} Supply_T$$

where X is the set of common events shared between them.

The MTR_T process has two main responsibilities. The first is to maintain a state reflecting which foreground values of type T are known by which trustworthy processes in the network at all times. This is achieved by synchronising upon all the relevant *send*, *receive* and *close* events in the network, where fresh values of type T are either introduced, gained or forgotten respectively. The second responsibility is to trigger the recycling of foreground values in the intruder's memory as soon as such a value becomes *forgotten* in the network. It is straightforward to determine when a foreground value v becomes *forgotten* by synchronising upon all the close events; as soon as the last trustworthy process storing v performs a close event, the manager will trigger the recycling process of v in the intruder's memory (through appropriate synchronisations). This is discussed in more detail later on when we present the extensions to the intruder component.

The second component defining the $Manager_T$ process is $Supply_T$, responsible for determining which fresh values of type T are supplied when requested by a trustworthy process. This is accomplished by synchronising upon all the *send* events in the network that require one or more fresh values of type T.

Example 4.1 *Consider the corrected NSPK protocol example introduced earlier and assume the system consists of one instance of A and one of B with identities $Alice$ and Bob respectively. The Nonce Manager process will synchronise with the network upon the following four categories of events:*

1. *All "send" events of message 1's by $Alice$ and message 2's by Bob. The nonce manager will determine which fresh nonce values are assigned to n_a (in the message 1's) and n_b (in the message 2's).*

2. *All "receive" events of message 1's by Bob and message 2's by Alice, where n_a and n_b are bound to fresh nonce values respectively. The manager is not interested in the case where Alice or Bob receives stale nonces, since this does not affect the recycling mechanism of the fresh ones.*

3. *All "close" events performed by $Alice$ and Bob, thereby signalling when they forget previously stored fresh values.*

4. *Events shared with the intruder for the purposes of triggering the recycling of fresh nonces in the system (discussed in more detail below).*

■

In the implementation of these manager processes, we parallelise the MTR_T component for efficiency purposes during the compilation phase. This component breaks down into a parallel structure quite naturally, since the monitoring of each fresh nonce is largely independent of one another. Using our adapted NSPK protocol example, assume the set of fresh nonces is define as $\{N_1, N_2\}$. In this case, we model the MTR_N process as the following parallel composition:

$$MTR_N \quad \widehat{=} \quad IndivMTR_N(N_1) \;{}_X\|_Y\; IndivMTR_N(N_2)$$

where N represents the type for nonces and X and Y represent the alphabets of the processes $IndivMTR_N(N_1)$ and $IndivMTR_N(N_2)$ respectively.

A further optimisation we use is based on the following observation: If a manager has several values it can hand out when asked for a fresh nonce, then clearly it does not matter which it gives, because all are treated completely symmetrically by the rest of the network. Therefore (i) there is no point in investigating the effects of handing out different options and (ii) there may be scope for choosing *which* value to hand out with a view to cutting down the overall range of states visited. The concept of symmetry reductions over states spaces is well known to be related to data independence and is discussed, for example, in [ES95, ES96, ID96].

Generalising manager processes

Previously, in [RB99] (and in our adapted-NSPK protocol example), the manager processes were constructed with the assumption that the protocol models conformed to the property that a message contains at most one variable being *monitored* by the corresponding manager; we refer to such managers as *one-dimensional*. This assumption makes the synchronisations between managers and monitored processes more straightforward, because a manager process does not have to handle multiple values in one synchronisation, and the managers collectively do not have to synchronise with each other.

We generalise the implementation of our techniques and abolish this limiting property above, by introducing *multi-dimensional* managers, as illustrated in the following example.

Example 4.2 *Suppose the agent Alice is running some protocol and receives a message M in which she newly acquires two nonces and one key, where nonces and keys are data independent types:*

$$\ldots \rightarrow Alice : \{n_1,\ n_2,\ k_1\}_{k_2}\ ,$$

where the key k_2 is assumed to be known by Alice already. Our simple one-dimensional managers could not handle this case; we require multi-dimensional managers where each manager process deals with 'tuples' of values rather than a 'single' one.

■

In this example, we would have two manager processes, $Manager_N$ for nonces and $Manager_K$ for keys. $Manager_N$ would have to monitor every pair (n_1, n_2) of nonces in M where at least one of them is bound to a fresh value. Intuitively, this is quite straightforward; in practise, care is needed when constructing these manager processes to ensure that all the corresponding sub-processes monitoring the fresh values synchronise together allowing a manager to monitor multiple fresh values in a single synchronisation. Furthermore, in this example, the key manager will also be synchronising upon the network message M to monitor the fresh key values bound to k_1. Once again, care is needed to ensure that the synchronisations taking place between the two managers do not interfere or restrict either from controlling their individual variables.

4.1.3 Intruder process

The intruder process is extended to incorporate the recycling mechanism of fresh values of data independent types introduced above.

The manager processes trigger the recycling mechanism for some fresh value v by synchronising with the intruder upon the event $recycle.(v, b)$; this has the effect of replacing all instances of v with the background value b in all the messages that contains v in the intruder's memory. The intruder process is implemented in such a way as to perform these mappings as an atomic action (modelled as a series of internal events).

The strategy used for determining which background values the recycled values get mapped to is a design choice of the programmer. The strategy we use is as follows. For every data independent type T, two distinct background values B_k and B_u are defined. We then map all foreground values *known* to the intruder to B_k and all the values *unknown* to the intruder to B_u. For each fresh value v that is ready to be recycled, the corresponding manager process synchronises with the intruder upon the channels *knowit* and *notknowit* to determine whether v is known to the intruder or not. The intruder process is defined such that it can synchronise on the $knowit.v$ event precisely when v is in its knowledge set. This is a simple mapping strategy that appears to work for a broad class of protocol models.

4.1.4 Specification processes

The application of the data independence techniques and corresponding extensions discussed in this chapter so far, have been concerned with the *implementation* side of our refinement checks only; namely the actual CSP system modelling the protocols. In this section, we discuss the extensions required to the *specification* processes used to capture the security properties we are interested in; these include all the untimed specifications offered in Casper and summarised in Chapter 2. We will use the *Agreement* authentication specification as an example to illustrate the extensions required. In brief, *Agreement*(A, B, ms) specifies that if B thinks he has successfully completed a run of the protocol with A, then A has been previously running the protocol with B, satisfying the following conditions: i) Both agents agreed as to which roles they took, ii) they both agree as to the values in ms, and iii) there is a one-to-one relationship between the runs of A and those of B. In terms of the corresponding signal events and using our adapted NSPK protocol example, the specification *Agreement*$(A, B, [n_a, n_b])$ is modelled as the following CSP process:

$$\begin{array}{l} \mathit{Spec}_0(a) \ \widehat{=} \\ \quad \mathit{signal} \,.\, \mathit{Running} \,.\, \mathit{Initiator\_role} \,.\, a?b?n_a?n_b \ \rightarrow \\ \quad \mathit{signal} \,.\, \mathit{Commit} \,.\, \mathit{Responder\_role} \,.\, b \,.\, a \,.\, n_a \,.\, n_b \ \rightarrow \ \mathit{STOP} \end{array}$$

This process is then instantiated accordingly to form the specification used in the refinement check against the protocol model. For example, if there is one instance of the initiator role with identity *Alice* that is able to perform two sequential runs, then the specification is defined as follows:

$$\mathit{Spec} \ \ \widehat{=} \ \ \mathit{Spec}_0(\mathit{Alice}) \ ||| \ \mathit{Spec}_0(\mathit{Alice}) \qquad (4.1)$$

When extending these specifications to incorporate the data independence techniques, there are two factors to be addressed. The first is that agents no longer perform a fixed number of runs; thus the specification would not be composed of a fixed number of instantiated processes interleaved as is the case above. The second point is concerned with handling the transformations upon fresh values as a result of the recycling mechanism. We address each of these factors in turn.

The first issue is concerned with agents being able to perform unbounded sequential runs. Extending the specification processes to handle this is straightforward and is modelled with an additional parameter X added to the process state; X is a set storing 3-tuples (b, n_a, n_b), one for each *signal.Running* event that A performs. In our CSP models that use the data independence techniques, we model the *Agreement* specification as the CSP process presented below.

$$
\begin{array}{l}
\mathit{Spec}_0(a, X) \mathrel{\widehat{=}} \\
\quad (\mathit{signal} \,.\, \mathit{Running} \,.\, \mathit{Initiator\_role} \,.\, a?b?n_a?n_b \;\rightarrow \\
\qquad \mathit{if}\ \mathit{Fresh}(\{n_a, n_b\})\ \mathit{then}\ \mathit{Spec}_0(a, X \cup \{(b, n_a, n_b)\}) \\
\qquad \mathit{else}\ \mathit{Spec}_0(a, X)) \\
\quad \Box \\
\quad (\Box (b, n_a, n_b) \,:\, X \;\bullet \\
\qquad \mathit{signal} \,.\, \mathit{Commit} \,.\, \mathit{Responder\_role} \,.\, b \,.\, a \,.\, n_a \,.\, n_b \;\rightarrow \\
\qquad\quad \mathit{Spec}_0(a, X - \{(b, n_a, n_b)\})) \\
\quad \Box \\
\quad \mathit{replace}?(x, y) \;\rightarrow \\
\qquad \mathit{Spec}_0(a, \{(b, n_a, n_b) \mid (b, n_a, n_b) \in X,\ n_a \neq x,\ n_b \neq x\})
\end{array}
$$

In the process definition above, we only store the values bound to b, n_a and n_b for a given *signal.Running* event in the case where those bound to n_a and n_b are fresh (verified by the function *Fresh*). We believe that trustworthy agents should not agree to commit to a run with one another based on *stale* values; this process will ensure us that if our protocol model does allow a trustworthy agent to commit under these circumstances, then our refinement will fail, alerting us to this problem. With regards to the *Signal.Commit* events, our specification only allows those whose bindings are taken from X.

The second factor to be considered is the effect of applying the collapsing mechanism for recycling fresh values, upon the specifications. As discussed in Chapter 3, we must be careful that the application of collapsing functions upon the specifications does not weaken them and cause false positives. To ensure this, such specifications must satisfy the two conditions in Proposition 3.2 (Page 62). To recall, consider a specification S. For every state $S_n(X)$ of S, if a collapse ϕ can occur (represented in our models by the replace event) leading to a new state $S_m(X')$, then the following two conditions must hold: (a) The set of traces of $S_m(X')$ is a subset of $S_n(X)$ (and therefore the effect of the collapse (if anything) makes S more stringent); and (b) X' contains no value in T which is the image under ϕ of more than one thing. This latter condition ensures that any equalities existing *after* the collapse between values in the states of S and the implementation-side of the refinement (in our case the protocol model), also held *before*.

We ensure that specifications involving the collapsing mechanism satisfies these two conditions by removing all instances of a fresh value v when v

gets recycled. For any given state $S_n(X)$ of a specification S, the collapse of v (represented by the replace event in the CSP process) will lead to a state $S_m(X')$, where $X' \subseteq X$ and therefore the resulting set of traces of $S_m(X')$ will be a subset of the traces of $S_n(X)$. Furthermore, by removing all instances of a recycled value, we also ensure that condition (b) is satisfied (since the removal of these values will not change the existing equalities).

As illustrated in the example above, our specification process must therefore synchronise upon the *replace.*(x, y) events (representing the recycling of the fresh value x to the stale value y). Once such an event has occurred, we remove all tuples containing the value x from the set X in the process state.

Using the same example as we did for the specification in (4.1), if there is one instance of the initiator role with identity *Alice* (able to perform an unbounded number of sequential runs), then we instantiate the specification process as follows:

$$Spec \;\widehat{=}\; Spec_0(Alice, \{\})$$

The extensions made to the other specifications offered by Casper are similar to those discussed here.

4.2 State space reduction strategies

Whilst extending our CSP models to incorporate the data independence techniques, it soon became apparent that these techniques cause a substantial increase in the state space required per protocol model, than was previously the case without them. For this reason, we developed a series of effective state space reduction strategies that address this problem. In this section, we discuss two of these strategies and the degree of improvement they provide (a third strategy is presented in Chapter 5).

4.2.1 Strategy 1: Optimising Casper-generated scripts

The first optimisation we made concerned the basic design of the CSP models generated by Casper. As discussed in more detail in Chapter 2, these models make use of *signal events* that reflect the state of mind of the trustworthy nodes running the protocol and are then used to specify security properties. For example, consider our adapted NSPK protocol example, where *Alice* and

Bob plays the initiator and responder role respectively. Suppose we want to test whether *Alice* is authenticated to *Bob*, and whether they agree upon the values bound to the nonces n_a and n_b. In this case, we would have the following two signal events in our protocol model[1]:

1. $Signal.Running.Alice.Bob.n_a.n_b$ reflecting the fact that *Alice* believes she is running the protocol with *Bob*, agreeing upon the values bound to n_a and n_b.

2. $Signal.Commit.Bob.Alice.n_a.n_b$ reflecting the fact that *Bob* believes he has completed a protocol run with *Alice* agreeing upon the values bound to n_a and n_b.

Previously, the definitions of the trustworthy nodes generated by Casper would interleave these signal events with the events representing the messages of the protocol. These events did not take part in the actual protocol run itself (visible to the environment only), and were used solely to flag the current state of mind of the corresponding processes. This method is a simple and effective way of representing the signal events required for specifications, since it does not interfere with the actual flow of messages and the synchronisations within the network. Unfortunately, these additional signal events greatly increases the number of traces of each process, and so greatly increases the state space. To overcome this problem, we redesigned our models so that instead of interleaving them with message events, the signals are introduced at the top most level, via an appropriate renaming of message events of the overall system process. Using the same example above, rather than explicitly add these signal events to the initiator and responder processes, we rename the corresponding messages at the system level. In this case, the *sending* of message 3's by *Alice* is renamed to her *running* signal events; similarly, the *receiving* of message 3's by *Bob* is renamed to his *commit* signal events.

A problem arises, however, if the message event does not contain all the information needed to construct the corresponding signal event. This is illustrated in the following example.

[1]We are simplifying the structure of the signal events slightly for ease of presentation.

Example 4.3 *Consider our adapted NSPK protocol example. As described above, the commit event of the responder agent Bob is associated to the* receiving *events of message 3's. In our CSP models, these events take the following form:*

```
(Msg3, Encrypt.(PK.(Bob), <nb>))
```

However, the signal event also needs the value bound to n_a *at this point, which is not provided in this message construct.*

■

To overcome this problem, we extended the CSP data structure representing the protocol messages by adding an additional field containing all the other information required; this field has *no* influence on the flow of messages in the protocol runs. In Example 4.3, the message 3 corresponding to the desired signal event becomes:

```
(Msg3, Encrypt.(PK(Bob), <nb>), <na>)
```

where the extra field `<na>` completes the set of required variables for constructing our signal event.

This optimisation on Casper has led to a dramatic reduction in the state space sizes of the protocols we model and check using FDR. For example, ignoring the use of the data independence techniques, the number of states explored for our adapted NSPK protocol example (with two instances of the initiator and two of the responder agents running in parallel) using the new version of Casper is 15,050 states as opposed to the old version requiring 425,734 states; the value for the new version incorporates an optimisation inspired by and similar to one of those described in [SS98], namely that the intruder never acts when a trustworthy agent can output. Without this degree of optimisation, most protocol models would prove to be infeasible to run with the incorporation of the data independence techniques.

4.2.2 Strategy 2: Stale knowledge

The second state space reduction strategy is applicable to the CSP models that incorporate the data independence techniques only. When applying these techniques, we quickly ran into problems with the size of the state space of the protocol model. One reason for this is the gradual build up within the intruder of knowledge gathered from past protocol runs. All these messages have only background values in the places occupied by the data independent types, because all foreground values have been mapped to them by the recycling mechanism.

If there are N such *stale* messages that the intruder might gain, then the effect is to multiply the total state space by as much as 2^N. To eliminate this problem, we observe that eventually the intruder can be expected to gain a complete set of this *stale knowledge* (with all values of data independent types mapped to background values), and can therefore exploit any such stale knowledge when attempting to perform attacks upon the protocol. So we simply calculate, at the start of a run, what the eventual set of stale knowledge ought to be and give this to the intruder as part of his initial knowledge. If done properly, this completely eliminates the state space explosion problem presented above.

In the case studies presented in [RB99], these sets of stale knowledge were calculated by hand. However, this proved to be error prone. Therefore, we have automated the calculation of the stale knowledge by performing a series of *pre-run simulations* between combinations of agents, including cases where the intruder takes on some of the roles, taking care to distinguish between the values of data independent types known and unknown by the intruder included within the messages. These simulations do not affect the state space, since we do not perform any actual runs of the protocol. Instead, we make use of the existing computational methods that are used to calculate the intruder's initial knowledge from initially known values specified by the user, closed under the deduction rules. The simulations are performed by taking the set of messages that the intruder would see by observing a typical run, using the mechanism outlined above to calculate the closure under the deduction rules and then mapping fresh values of data independent types to appropriate background values. Calculating these sets based on the model of the intruder allows us to automatically capture any subtleties within the intruder's specified behaviour.

Example 4.4 *Consider our adapted NSPK protocol example (presented on Page 65), with agents Alice and Bob playing initiator and responder roles respectively. We define $Nonce_B$ to be the set of background (stale) nonces that all the fresh nonces will be recycled to, as follows:*

$$Nonce_B \quad = \quad \{Old_P,\ Old_S\}$$

where all fresh values known to the intruder at the point of recycling will be mapped to Old_P, and all those unknown to him will be mapped to Old_S.

In this example, there are 8 simulations (representing old runs) to consider:

1. *Alice (as role A) with Alice (as role B);*
2. *Alice (as role A) with Bob (as role B);*
3. *Alice (as role A) with the intruder (as role B);*
4. *Bob (as role A) with Bob (as role B);*
5. *Bob (as role A) with Alice (as role B);*
6. *Bob (as role A) with the intruder (as role B);*
7. *The intruder (as role A) with Alice (as role B);*
8. *The intruder (as role A) with Bob (as role B).*

Each of these generate 3 stale messages. For example, the stale message set resulting from an old protocol run between Alice (as role A) and Bob (as role B) is:

$$\{\{Old_S,\ Alice\}_{PK(Bob)},\ \{Old_S,\ Old_S,\ Bob\}_{PK(Alice)},\ \{Old_S\}_{PK(Bob)}\}$$

Notice that the only background nonce present in this set of stale messages is Old_S. This reflects the fact that the intruder does not learn the fresh nonces introduced by Alice and Bob, by simply observing a complete run of the protocol between them. On the other hand, the stale message set resulting from an old protocol run between Alice (as role A) and the intruder (as role B, using identity Mallory) is:

$$\{\{Old_P,\ Alice\}_{PK(Mallory)},\ \{Old_P,\ Old_P,\ Mallory\}_{PK(Alice)},\\ \{Old_P\}_{PK(Mallory)}\}$$

since, the intruder uses the background nonce Old_P as his nonce (that he introduces in message 2) and learns the fresh value introduced by Alice in message 1. This value is therefore subsequently mapped to the known background value Old_P when recycled.

The total number of stale messages generated by this system is 24. This has the potential of increasing the total state space by a multiple of 2^{24}.

■

4.3 Automating data independence techniques

Initially, we implemented the data independence techniques within our CSP protocol models by hand, carefully crafting them to suit each case study. However, this process proved to be time-consuming, error-prone and required a substantial knowledge of the CSP language and a good understanding of the theory underlying the data independence techniques; this, in turn, limited to use of these methods for a broader class of users interested in analysing security protocols.

Therefore, we have automated these techniques by incorporating them into the Casper compiler (introduced in Chapter 2). In this section, we discuss the extensions made to Casper and give an example of an input script using these techniques; we use our adapted NSPK protocol example that we have been referring to throughout the chapter. This work has proved to be very useful for generating our CSP protocol models, since it only takes Casper a couple of seconds to perform the task, which in the past took us several weeks per protocol model.

4.3.1 Extending Casper input script

Our aim was to abstract away the design and complexity of the data independence techniques from the users of Casper, making them much more accessible. As we will see in our example script, the differences in the input script are minimal. Figure 4.2 presents the input script for our adapted NSPK protocol example, and illustrates the main extensions made to incorporate the data independence techniques. We discuss each of them in turn.

Firstly, the user must indicate which data types are to be treated as data independent. Within the `#Processes` section, variables of such types are indicated using the `generates` keyword: values for such variables will be freshly supplied by the corresponding manager processes. In our example in Figure 4.2, nonces are regarded as data independent, so the corresponding variables `na` and `nb` are indicated as being generated in this way; by contrast, without the use of data independence techniques, these variables were introduced as parameters of the processes, with the parameters being instantiated in the `#System` section.

In the `#Actual Variables` section, we declare the actual variables which will be used in our actual system. We now also need to classify all vari-

```
-- Casper input script for the corrected NSPK protocol.

#Free Variables
A, B : Agent
na, nb : Nonce
SK : Agent -> SecretKey
PK : Agent -> PublicKey
InverseKeys = (PK,SK)

#Processes
INITIATOR(A) knows SK(A) generates na     Values for na and nb will be freshly supplied
RESPONDER(B) knows SK(B) generates nb     by the Nonce Manager processes.

#Protocol description
0.    -> A : B
1.  A -> B : {na, A}{PK(B)}
2.  B -> A : {na, nb, B}{PK(A)}
3.  A -> B : {nb}{PK(B)}

# Specification
Agreement(B, A, [na, nb])

#Actual Variables                          Source of values used by the Nonce Manager to
Alice, Bob, Mallory : Agent                generate a continuous supply of fresh nonces.
N1, N2, N3 : Nonce (Foreground)
NBs : Nonce (UnknownBackground)            Background nonces used for recycling purposes.
NBp : Nonce (KnownBackground)

#Functions
symbolic PK, SK

#System
INITIATOR(Alice)
RESPONDER(Bob)                             Optional:  User can chose whether agents
[WithdrawOption = True / False ]                      can withdraw at any point
                                                      during a run or not.
#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, NBp, SK(Mallory), PK}
[StaleKnowledge = True / False ]           Optional: Use of state space reduction strategy.
```

Figure 4.2: Casper input script for the adapted NSPK protocol example, incorporating the data independence techniques.

ables of data independent types as either `Foreground`, `KnownBackground` or `UnknownBackground` values.

In the case of the foreground values, the user needs to estimate how many values the manager will need[2]; if too few values are given, FDR will give a trace error, indicating that the manager process ran out of fresh values, and so was unable to allocate another value; in this case, the user can simply edit the Casper script to include an extra value. It is advisable to declare the minimum necessary number of foreground values for each type, since increasing this number will cause the state space of the system to grow dramatically.

For each data independent type, the user must declare exactly one value as a `KnownBackground` value, and one as a `UnknownBackground` value. These are used as the background (*stale*) nonces for the recycling mechanism of all the fresh values; we use the mapping strategy described in the previous section, namely, all fresh values known to the intruder at the point of recycling are mapped to the *known* background value, and all those unknown to him are mapped to the *unknown* one. In our example, these are the values *NBp* and *NBs* respectively.

In the `System` section, we declare the actual system to be verified. We no longer instantiate fresh variables of data independent types, such as nonces and session keys, since they will be freshly supplied by the managers. Also, since we now have the assumption that all the declared participants can perform an unbounded number of sequential runs, it suffices to declare each participant once.

A further extension concerns the ability for the agent processes to withdraw *during* a session or not, captured by `WithdrawOption = True / False`. For full generality, one should allow agents to withdraw, but this does tend to increase the state space, so we make it an option.

In the `Intruder Information` section, we add all the *known* background values to the intruder's initial knowledge set; this is important for the recycling mechanism we use to ensure that the intruder is not weakened by any of the transformations we perform upon the fresh values. In our example, this refers to the nonce *NBp*.

[2] We do not currently have any heuristics for computing the minimum number of foreground values the manager will need. At present, we estimate this for each case and then supply the managers with more if requested by FDR. Since these models are generated automatically by Casper, adding and removing fresh values is very simple.

The user also has the option of supplying the intruder with the stale knowledge as part of his initial state. As discussed above, this is a very effective state space reduction strategy and is a highly recommended option. The main reason why one might wish to disable this option, is in the cases where attacks are found. It is sometimes the case that an attack is less clear when using the stale knowledge, since the user is not provided with all previous runs required (as this is the assumption provided by the stale knowledge). Under these circumstances, it can be useful to disable this option, thereby finding a *whole* attack with all the necessary runs.

4.4 Example: Analysing the Otway-Rees protocol

In this section, we illustrate how the data independence techniques are applied to the Otway-Rees protocol and discuss the effects the state space reduction strategies – in particular the use of stale knowledge – have upon the analysis and the state space. The version we use is essentially the same as the one presented in [BAN89], defined as follows:

$$\begin{array}{lll}
\text{Message 1.} & A \rightarrow B & : \ n_a \\
\text{Message 2.} & A \rightarrow S & : \ \{n_a,\ A,\ B\}_{SKey(A)} \\
\text{Message 3.} & B \rightarrow S & : \ n_a,\ n_b \\
\text{Message 4.} & B \rightarrow S & : \ \{n_a,\ A,\ B\}_{SKey(B)} \\
\text{Message 5.} & S \rightarrow B & : \ n_a,\ \{n_b,\ k_{ab}\}_{SKey(B)} \\
\text{Message 6.} & S \rightarrow A & : \ \{n_a,\ k_{ab}\}_{SKey(A)}
\end{array}$$

The only difference from the version in [BAN89] is that all the messages are sent directly to their destinations, rather than passing through intermediate agents; redirecting messages in this way does not effect security (Hui and Lowe in [HL01]). This particular version has a number of well known authentication flaws that we will use to illustrate our methods.

4.4.1 System being verified

We verify the following system, as presented in the corresponding Casper input script for this example:

Initiator(Alice, Sam)
Responder(Bob, Sam)
Server(Sam)

where *Initiator* represents the role A, *Responder* represents the role B and *Server* represents the role S.

We declare the types *Nonce* (types of n_a and n_b) and *SessionKey* (type of k_{ab}) as data independent. Hence our system consists of one instance of each existing role, where they can each perform an unbounded number of sequential runs; the manager processes will be responsible for supplying fresh values for n_a, n_b and k_{ab} when required. We declare the set of available fresh nonces and keys as:

$$\begin{aligned} \mathit{Nonce}_F &= \{N_1,\ N_2,\ N_3\} \\ \mathit{SessionKey}_F &= \{K_1,\ K_2,\ K_3\} \end{aligned}$$

and the known and unknown background (stale) nonces and keys as:

$$\begin{aligned} \mathit{KnownBackground}\,(\mathit{Nonce}) &= \mathit{NB}_p \\ \mathit{KnownBackground}\,(\mathit{SessionKey}) &= \mathit{KB}_p \\ \mathit{UnknownBackground}\,(\mathit{Nonce}) &= \mathit{NB}_s \\ \mathit{UnknownBackground}\,(\mathit{SessionKey}) &= \mathit{KB}_s \end{aligned}$$

4.4.2 Resulting analysis

We verify our system for the following Casper authentication specification:

$\mathit{Agreement}\,(B,\ A,\ [\,])$

This specifies that if A thinks she has successfully completed a run of the protocol with B, then B has previously been running the protocol, apparently with A, under the following conditions: i) Both agents agreed as to which roles they took, and ii) there is a one-to-one relationship between the runs of A and those of B.

Without the use of stale knowledge, the following authentication attack is found:

(1) Message 1. $Alice \rightarrow I_{Alice} : N_1$

(2) Message 2. $Alice \rightarrow Sam : \{N_1,\ Alice,\ Alice\}_{SKey(Alice)}$

(3) Message 3. $I_{Alice} \rightarrow Sam : N_1,\ N_1$

(4) Message 4. $I_{Alice} \rightarrow Sam : \{N_1,\ Alice,\ Alice\}_{SKey(Alice)}$

(5) Message 5. $Sam \rightarrow I_{Alice} : N_1,\ \{N_1,\ K_1\}_{SKey(Alice)}$

(6) Message 6. $Sam \rightarrow Alice : \{N_1,\ K_1\}_{SKey(Alice)}$

(7) *Sam* withdraws from this run as *Server*, thereby forgetting the fresh values N_1 and K_1.

(8) K_1 is recycled to KB_s in the intruder's memory and is therefore regarded as fresh again.

(9) *Alice* withdraws from this run as *Initiator*, thereby forgetting the fresh value N_1.

(10) N_1 is recycled to NB_p in the intruder's memory and is therefore regarded as fresh again.

(11) Message 1. $Alice \rightarrow I_{Bob} : N_1$

(12) Message 2. $Alice \rightarrow I_{Sam} : \{N_1,\ Alice,\ Bob\}_{SKey(Alice)}$

(13) Message 2. $I_{Alice} \rightarrow Sam : \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)}$

(14) Message 3. $I_{Alice} \rightarrow Sam : NB_p,\ N_1$

(15) Message 4. $I_{Alice} \rightarrow Sam : \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)}$

(16) Message 5. $Sam \rightarrow I_{Alice} : NB_p,\ \{N_1,\ K_1\}_{SKey(Alice)}$

(17) Message 6. $I_{Sam} \rightarrow Alice : \{N_1,\ K_1\}_{SKey(Alice)}$

On receiving the final message 6 in step (17), *Alice* believes she has completed a run of the protocol, taking role *Initiator*, with *Bob* (as the *Responder*). The truth is that *Bob* was never actually running the protocol; the intruder successfully impersonated him as playing the *Responder* role.

This attack relies on a *previous* run, represented by the messages in step (1) through to step (6), between *Alice* playing the role as the *Initiator* and the intruder pretending to be *Alice* as the *Responder*. The intruder then uses the *old* message 2 to attack the protocol in the second run, by replaying it in step (13) as the message from *Alice* (as *Initiator* of the second run). Notice that this message takes the form

$$\{NB_p,\ Alice,\ Alice\}_{SKey(Alice)}$$

since the fresh nonce N_1 in the first run was recycled to NB_p.

If we analyse this system without the data independence techniques and, for example, declaring a single instance of each role in the system, each only being able to perform one run, then we do not find this attack. Clearly, the more experienced and thorough Casper user would probably have considered other system instantiations to analyse, in which case they would have found the flaw. However, it does emphasise the usefulness of our techniques providing the assumption that all the instantiated roles can perform an unbounded number of runs each (sequentially).

An interesting observation is that the attack presented above was found in 801,919 states; this does not really provide much confidence regarding the feasibility of our work when looking at larger examples or being able to complete a proof that no attack exists (thereby searching the whole state space without any counter-examples). This model had already incorporated the first state space reduction strategy, namely the optimisation of the Casper models we presented in Section 4.2 - without this, finding the attack upon this model (using data independence) would be infeasible.

We now consider the same model described above, but this time we incorporate our second state space reduction strategy, namely the use of stale knowledge (introduced in Section 4.2).

The following (equivalent) attack is found:

$$\begin{array}{lrcl}
\text{Message 1.} & Alice \rightarrow I_{Bob} & : & N_1 \\
\text{Message 2.} & Alice \rightarrow I_{Sam} & : & \{N_1,\ Alice,\ Bob\}_{SKey(Alice)} \\
\text{Message 2.} & I_{Alice} \rightarrow Sam & : & \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 3.} & I_{Alice} \rightarrow Sam & : & NB_p,\ N_1 \\
\text{Message 4.} & I_{Alice} \rightarrow Sam & : & \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 5.} & Sam \rightarrow I_{Alice} & : & NB_p,\ \{N_1,\ K_1\}_{SKey(Alice)} \\
\text{Message 6.} & I_{Sam} \rightarrow Alice & : & \{N_1,\ K_1\}_{SKey(Alice)}
\end{array}$$

Once again, on receiving the final message 6, *Alice* believes she has completed a run of the protocol, taking role *Initiator*, with *Bob* (as the *Responder*).

This attack is precisely the same as the previous one we found above. The main difference is that, by using the stale knowledge, the system already assumes that all previous runs between existing agents have taken place. For example, the old message 2 replayed by the intruder to *Sam* was known to him as part of his initial knowledge, whereas previously, the system had to go through a complete run before it was generated. This is enormously beneficial in terms of the state space, since FDR no longer has to consider these previous runs. This attack was found in 7,747 states, illustrating this fact. In Chapter 5, we introduce a third state space reduction strategy that cuts the state space for finding this attack down even further to 168 states.

4.5 Discussion

In this chapter, we have presented the design and implementation of the data independence techniques introduced in Chapter 3, within our CSP protocol models. These techniques allow us to prove far more complete results than hitherto, since agents are able to perform an unbounded number of sequential runs each.

Section 4.2 introduced two highly effective state space reduction strategies, thereby allowing our methods to be applied to larger and more complex protocols, and addressing the increase in state space that typically occurs when applying the data independence techniques. In Section 4.4, we illustrated the effects of our state space reduction strategies by applying them to the Otway-Rees protocol; the improvement was of 3 orders of magnitude (including both reduction strategies). Strategies such as these are particularly important in the model checking world, as the problem of state space explosion is a common phenomenon when addressing large problems such as security protocols with many agent runs.

The implementation process of the data independence techniques was fully automated by extending incorporating them into Casper. Our aim was to abstract away the design and complexity of the data independence techniques from the users of Casper; this has indeed been achieved as discussed in Section 4.3 and illustrated by the example input script in Figure 4.2. This required a substantial amount of work within Casper in order to generate the necessary manager processes, recycling mechanism of all the fresh values within the managers and the intruder process, and the specification processes accordingly.

Chapter 5

Internal agents

In Chapters 2 and 4, we described the design of our CSP protocol models with and without the use of the data independence techniques. We saw how, in both cases, participating agents and servers are typically modelled as individual processes defined according to their role in the description of the protocol in question. These processes are then placed in parallel with the intruder (and manager processes in the latter model description) and communicate with one another by synchronising upon their corresponding *send* and *receive* messages (through the intruder). The advantage of this approach is that it is a very simple and intuitive way of modelling agents in CSP. On the other hand, the disadvantage is that the number of instances of agents you can model is finite. This means that if we do not find an attack upon a given protocol model, this does not prove that none exists for a higher degree of parallelism. Furthermore, the state space grows so quickly for each agent process added, that it is only feasible to perform refinement checks for a small number of agents in parallel (especially in the models that use the data independence techniques).

In this chapter, we present an alternative approach to modelling agent processes within our CSP models. This involves capturing the functionality of participating nodes within the intruder as a series of deductions, instead of modelling them as individual CSP processes. The initial motivation for this work was as a state space reduction strategy; this approach has indeed proved to be very effective for this purpose and together with the state space reduction strategies presented in Chapter 4, form an important part of our CSP models that use the data independence work. However, a more recent motivation for this work is to enable our CSP models to capture attacks upon protocols requiring a much higher degree of parallelism within a single agent identity than previously feasible using the FDR model checker. Chapters 6 describes how we achieve this; furthermore, for certain classes of protocols, we show how these extended models capture attacks requiring *any* degree of parallelism, still within the confines of a finite refinement check.

This chapter is arranged as follows. In Section 5.1, we present the design and implementation of capturing agents internally within the intruder component; we refer to them as *internal agents*. Our initial work [RB99] involved internalising roles that were not part of the secrecy or authentication specifications being verified, since this avoided having to address the question: What is the state of mind of a participant that has been internalised and how do we construct the necessary signal events accordingly? Typically, server-type roles fall into this category.

Internalising agents can sometimes increase the possibility for false attacks. This is due to the fact that we capture internal agents' functionality as deductions and generations within the intruder, and if we are not careful, allow the intruder to perform these extra deductions whenever he wants, even when it does not actually reflect the true behaviour of the internal agents in question. This problem proves to be uncommon when internalising the server roles in the initial stages of this work; however, it can sometimes arise when internalising more general agent roles. We discuss this issue in Section 5.2.

The modelling techniques we have presented thus far for analysing security protocols have been automated by incorporating them into Casper, making them accessible to a broader class of users. Section 5.3 presents an overview of this work. We believe that this is important, since these sorts of modelling techniques require a high level of skill in a specialised area, rendering them limited in use otherwise. Making use of this extended version of Casper, we present some examples in Section 5.4 illustrating how our new approach of internalising agents applies in practice and the beneficial effect it has upon the state space.

In Section 5.5, we address the question raised above, of how one could capture signal events from internal agents and illustrate this with an example in Section 5.6. Given the benefits of internalising agents, this is a useful extension to our framework for analysing security protocols.

5.1 Internal agents

In this section, we introduce a new approach to modelling protocol agents, referred to as *internal agents*, and present their design within our CSP protocol models.

An *internal agent* is one whose functionality is modelled as a series of special deductions that the intruder can perform in addition to his own standard set of deductions, instead of modelling them as individual CSP processes. By internalising an agent A into the intruder, we do not allow the intruder to gain access to the state of mind of A; our aim is to construct internal agents in such a way that the intruder is given the ability to behave as though they are *oracles* that he can use. For example, consider such a deduction that generates a message M, encrypted under the public key of some trustworthy agent within the system (and unknown to the intruder). In this case, the intruder only gains the knowledge of M as a whole, not any of the sub-components within the encryption.

We classify internal agents into two categories. The first contains those that do not introduce any fresh variables of a data independent type at any point during the protocol. Internal agents in this first category are straightforward to incorporate into the intruder process and are captured with appropriate deductions. The second category are those internal agents that introduce fresh variables of data independent types at some stage in the protocol runs. Examples in the latter category include internal servers and agents that introduce fresh session keys and nonces respectively.

Deductions performed by the intruder are usually modelled by pairs of the form (X, f) (also written as $X \vdash f$), where X is a finite set of facts and f is a fact that it can construct if it knows the whole of X; for example, $(\{\{M\}_K, K\}, M)$ means that if the intruder knows message M encrypted with key K and he knows K, then he can deduce M. The functionality of the first sort of internal agent can be captured with this type of deduction within the intruder: we get a deduction (X, f) if, after the agent is told the messages in X, it can be expected to emit f (where f will be functionally dependent on X). A typical example of this kind is the server in the TMN protocol [TMN90], whose function was to receive two messages M_1 and M_3 and construct a corresponding third message M_4, where M_4 only contains variables in M_1 and M_3 (thereby not introducing any fresh variables into the system). Thus, the functionality of the server is captured internally by all valid instantiations of the following deduction:

$$\{M_1,\ M_3\} \ \vdash \ M_4$$

For internal agents that fall into the second category (those that introduce fresh values), these standard deductions do not suffice. The problem is that they cannot capture the fact that fresh values are supplied and controlled by their respective manager processes. Consider the following example.

Example 5.1 *Consider a protocol with the following first two messages:*

$$\begin{array}{ll} \text{Message 1.} & A \rightarrow S \ : \ \{n_a\}_{SKey(A)} \\ \text{Message 2.} & S \rightarrow A \ : \ \{k_{ab},\ n_a\}_{SKey(A)} \end{array}$$

where n_a is a nonce freshly introduced by agent A, k_{ab} is a key freshly introduced by the server S and $SKey(A)$ is a secret long term symmetric key shared only by S and A.

If S is modelled as internal, then we need to capture the fact that S generates a message 2 based on the message 1 received, together with a fresh key. This cannot be captured by a standard intruder deduction of the form:

$$\{n_a\}_{SKey(A)} \ \vdash \ \{k_{ab},\ n_a\}_{SKey(A)}$$

since, how is a fresh key supplied and bound to k_{ab} and who makes this decision? We must enable the Key Manager to take part in this deduction and be responsible for the keys supplied.

■

It is therefore necessary to introduce a special sort of deduction, referred to as a *generation*. A *generation* has the form (t, X, Y) (also written as $t, X \ \vdash \ Y$), where t is a non-empty sequence of the fresh objects being created, X is a finite set of input facts and Y is the set of facts generated, which contain the fresh values that are produced. In Example 5.1 above, t, X and Y would consist of a single fresh key, a message 2 and a corresponding message 3 respectively:

$$\begin{array}{rcl} t & = & \langle k_{ab} \rangle \\ X & = & \{ \ \{n_a\}_{SKey(A)} \ \} \\ Y & = & \{ \ \{k_{ab},\ n_a\}_{SKey(A)} \ \} \end{array}$$

The sequence t then synchronises with the appropriate managers (in our example above the sequence would be of length 1 and synchronise with the Key Manager only) and the set Y is functionally dependent on X and t. In other words, if we know what messages went into our server, and what fresh values it generated in response, then we know what messages it will output.

5.1.1 Extending intruder's deductive system

In this section, we give an overview of the CSP design and implementation of internal agents. For the purposes of clarity, we will refer to the intruder's standard deduction set as D_I, the set of all deductions reflecting the functionality of internal agents as D_A (where no fresh values are introduced) and finally, the set of all generations reflecting the functionality of internal agents as G_A (where fresh values are introduced). As discussed above, the deductions in D_A are captured in precisely the same way as those in D_I. An obvious way to treat D_A would be to simply merge them with D_I; indeed, this is what we did for the server-type roles we considered initially and in the case studies presented in [RB99]. However, there are a number of reasons why this is not the best approach when considering internalising agent roles in general. We discuss each of these in turn below.

Firstly, we present some restrictions upon the deductions in D_A to address the issue of false attacks (in Section 5.2). If we merged the two sets, then we would have to be extremely careful when extracting those deductions upon which restrictions will be placed. It is important that we do not accidently restrict any intruder deductions in D_I. For example, merging all deduction sets into one could potentially lose information about any deductions that were both members of D_I and D_A (since the intersection of these two sets is not necessarily empty).

Secondly, we address the issue of constructing signal events for internal agents through the renaming of their corresponding deductions in D_A (Section 5.5). Once again, the design is less clear and therefore more error-prone in the case where we only have one deduction set; we want to be sure that we are only constructing signal events from those deductions corresponding to the internal agent's functionality in question.

Thirdly, all of the intruder's deductions (represented by the *infer* events) are chased using FDR's symmetric reduction function *chase* (introduced in Chapter 2). We cannot assume that all the internal agents' deductions can be chased, since some of them may fall into the two categories mentioned above and therefore could be involved in restrictions of some kind or used for the construction of signal events. A set of events E should only be chased when one can ensure that the outcome will always be the same irrespective of the ordering of the events in E. This is clearly not the case for a set of events that are restricted depending on the state of the system. Furthermore, deductions used for constructing signal events are renamed at the top level of the system definition. Hiding such events in order to apply the chase function is therefore not possible.

Finally, a less harmful yet still important point concerns interpreting the counter-example trace that FDR generates representing an attack. It is very useful to be able to distinguish between deductions the intruder has made himself versus those he has made on behalf of an internal agent. This is more difficult to identity if all the deductions are merged together as one set over the channel *infer*. Hence, for the purposes of clarity and correctness within our CSP protocol design, it seems appropriate to keep the two sets D_A and D_I separate.

We introduce a new channel *deduce* within the intruder upon which he can perform internal agent deductions. The type of *deduce* takes the same form to that of the standard *infer* channel (used for intruder's own deductions) and is defined as a set of pairs (X, F), where X is a finite set of facts and F is the set of facts that the corresponding internal agent could generate once it had received the whole of X. When more than one agent role is internalised, we also ensure that the corresponding deductions are kept separate. This is done for similar reasons as listed above, in particular for clarity within the design of the CSP model itself.

Example 5.2 *Consider a protocol defined to have an* Initiator *and a* Responder *role, both modelled as internal agents. In this case, we would extend our channel type of* deduce *to having an extra field, identifying the agent role for whom the deduction was performed. Therefore, if D_I and D_R represents the deduction sets of the* Initiator *and* Responder *roles respectively, then the intruder would be able to perform the following* deduce *events:*

$$Agent\_Deductions \quad = \quad \bigcup \left\{ \begin{array}{l} \{deduce \;.\; Initiator \;.\; d_i \quad | \quad d_i \in D_I\}\,, \\ \{deduce \;.\; Responder \;.\; d_r \quad | \quad d_r \in D_R\} \end{array} \right\}$$

■

In the case of the generation sets G_A within the intruder reflecting the functionality of the internal agents creating messages that contain fresh values, there are no problems concerning the intruder component itself; the intruder is not defined in terms of any "standard" generations. Therefore, no conflicts arise between the events representing the internal agents' generations and any other event the intruder is able to perform. The generations in G_A are performed upon the channel *generate*. In the event that we have more than one internalised agent, we implement the same division as done for the deduction sets. As is the case for the internal agents' deduction sets (and for the same reasons), we are not allowed to apply the *chase* function to any of the *generate* events.

5.1.2 Constructing internal agents

In this section, we present an informal overview of the general strategy we use for constructing the set of deductions and generations of the internal agents in our CSP protocol models. This will be followed by an example illustrating how we apply this strategy in practice.

In order to construct the deductions and generations for a given internal agent A, we step through the protocol description and consider all the messages involving the participation of A (either as the sender or receiver).

Once we have identified the relevant messages (preserving the original ordering in the protocol description) of a given agent A to be internalised, we start the process of constructing the deductions and generations that reflect the functionality of A as follows. We step through each message until we find the first message M where A is the sender (as opposed to the receiver) of M. At this point, we know that either a deduction or a generation is required to reflect the *creation* of M, based on all the previous messages involving A; these then form the antecedent of the resulting generation or deduction. Whether we need to construct a representative deduction or a generation is dependent on whether A introduces any fresh variables in M or not.

If A does not introduce any fresh variables in M, then we construct a corresponding deduction of the form:

$$\{\textit{All messages before M involving A}\} \;\vdash\; M$$

On the other hand, if A does introduce some fresh variables $v_1, \ldots, v_k$ in M, then we need to construct a generation to reflect this additional new information, taking the form:

$$\left(\begin{array}{l} \langle v_1, \ldots, v_k \rangle, \\ \{All\ messages\ before\ M\ involving\ A\} \end{array} \right) \vdash \quad M$$

However, unlike the deduction construction, we don't necessarily construct a generation for each single message sent by A that introduce fresh variables. Instead, once we have found such a message M, we gather all consecutive messages $M_1^{'}, \ldots, M_n^{'}$ that are *sent* by A. This is done by simply stepping through the messages following M and collecting them until we get to a message where A is no longer the sender. At this point, the messages collected will be added to M and a generation is constructed taking the form:

$$\left(\begin{array}{l} \langle x_1, \ldots, x_i \rangle, \\ \{All\ messages\ before\ M\ involving\ A\} \end{array} \right) \vdash \quad \{M, M_1^{'}, \ldots, M_n^{'}\}$$

where $x_1, \ldots, x_i$ are the fresh values introduced by A in $M, M_1^{'}, \ldots, M_n^{'}$.

We group these messages together because it simplifies the distribution of fresh variables when they are introduced and appear in several consecutive messages. It is a natural way of ensuring that, for a given fresh variable v that appears in several consecutive messages sent by A, the fresh *value* that is assigned to v is indeed the same for each instance of v. In addition, since we only group together consecutive messages sent by the same agent, there is no reason why they should not be generated in one step; the latter messages are not influenced by any further input than the first message of the group (since we only group together consecutive messages where each one is sent by A only).

Once we have constructed the appropriate deduction or generation, we move onto the next *unhandled* message in the sequence that is sent by A and perform the whole process again. By unhandled, we mean that the message has not already been represented as part of the previous generation.

5.1.3 Example: Applying our internalisation strategy

To clarify the process of internalising agents, we apply it to a simple example below. It should be noted that this example is contrived solely for the purposes of illustrating the strategy we use to model internal agents; it is not one that would be useful for any practical purposes.

Example 5.3 *Consider a protocol with the following sequence of messages:*

$$\begin{array}{lll} \textit{Message } 1. & A \rightarrow B & : \; n_a \\ \textit{Message } 2. & B \rightarrow S & : \; \{A,\, n_b,\, n_a\}_{SKey(B)} \\ \textit{Message } 3. & S \rightarrow A & : \; \{B,\, k_1,\, n_a\}_{SKey(A)} \\ \textit{Message } 4. & S \rightarrow B & : \; \{A,\, k_1,\, n_b\}_{SKey(B)} \\ \textit{Message } 5. & B \rightarrow S & : \; \{n_b\}_{k_1} \\ \textit{Message } 6. & S \rightarrow A & : \; \{B,\, k_2,\, n_a\}_{SKey(A)} \\ \textit{Message } 7. & A \rightarrow S & : \; \{Sec_A,\, k_2\}_{SKey(A)} \\ \textit{Message } 8. & S \rightarrow B & : \; \{A,\, Sec_A,\, k_2,\, n_a\}_{SKey(B)} \end{array}$$

where n_a and n_b are fresh nonces introduced by A and B respectively; k_1 and k_2 are fresh keys introduced by S; and Sec_A is a freshly introduced secret message of some other type T (neither a nonce nor a key).

■

In this example, there are three agents roles, namely A, B and S. We consider the process of internalising the server role S. The first step is to identify all messages that involve S either as the sender or the receiver; this sequence consists of all the messages starting from message 2 through to and including message 8.

The second step is to go through these messages and construct deductions and generations for the messages *sent* by S. The first of these is message 3. In this message, S introduces the fresh variable k_1 and therefore we know that we require a generation reflecting this fact. We then identify all the consecutive messages where S is the sender; together with message 3, these will form the *result* of this generation. In our example, there is one consecutive message, namely message 4. Hence, the generation for this group is defined as follows:

$$(Gen_{S_1}) \qquad \langle k_1\rangle,\ \{\{A,\, n_b,\, n_a\}_{SKey(B)}\} \ \vdash \ \left\{ \begin{array}{l} \{B,\, k_1,\, n_a\}_{SKey(A)}, \\ \{A,\, k_1,\, n_b\}_{SKey(B)} \end{array} \right\}$$

The next message sent by S that we need to consider is message 6. Once again, S introduces a fresh variable, namely the key k_2 and therefore we need to construct another generation reflecting this. In this case, there are no consecutive messages sent by S (since message 7 is received by S, not sent) and therefore our generation is defined as follows:

$$(Gen_{S_2}) \qquad \langle k_2 \rangle, \left\{ \begin{array}{l} \{A,\ n_b,\ n_a\}_{SKey(B)}, \\ \{B,\ k_1,\ n_a\}_{SKey(A)}, \\ \{A,\ k_1,\ n_b\}_{SKey(B)}, \\ \{n_b\}_{k_1} \end{array} \right\} \vdash \{\{B,\ k_2,\ n_a\}_{SKey(A)}\}$$

The final message sent by S is message 8. In this message, no fresh variables are introduced and therefore we can simply construct the corresponding deduction as follows:

$$(Ded_{S_1}) \qquad \left\{ \begin{array}{l} \{A,\ n_b,\ n_a\}_{SKey(B)}, \\ \{B,\ k_1,\ n_a\}_{SKey(A)}, \\ \{A,\ k_1,\ n_b\}_{SKey(B)}, \\ \{n_b\}_{k_1}, \\ \{B,\ k_2,\ n_a\}_{SKey(A)}, \\ \{Sec_A,\ k_2\}_{SKey(A)} \end{array} \right\} \vdash \{A,\ Sec_A,\ k_2,\ n_a\}_{SKey(B)}$$

Thus, the server role S is captured by two generation sets (Gen_{S_1} and Gen_{S_2}) and one deduction set (Ded_{S_1}) when internalised within the intruder component.

5.2 Introducing false attacks

In this section we look at the issue of false attacks that can arise in our CSP protocol models as a result of internalising agents. We start by identifying what properties are lost when an agent A is modelled internally within the intruder, compared to A being modelled as an external process. From this, we look at how we can re-establish these lost properties by placing certain conditions upon the intruder, thereby reducing the problem of false attacks.

Although the CSP protocol models that use our data independence techniques have the potential for introducing false attacks, their occurrence has proved to be uncommon. In particular, we have found that internal server-type roles are rarely the cause of false attacks within our models. By server-type, we are particularly referring to those whose sole purpose is to distribute fresh session keys to the network and share previously established long term keys (assumed to be secure) with each participant. However in the more general case, where we are interested in internalising agent roles, this problem can become more prominent. The following example illustrates how the intruder can *misuse* internal agent deductions, thereby leading to potential false attacks.

Example 5.4 *Consider the following protocol description:*

$$\begin{array}{llll} \text{Message 1.} & A \rightarrow S & : & \{B,\ i_a\}_{SKey(A)} \\ \text{Message 2.} & S \rightarrow B & : & n_s,\ \{A,\ i_a\}_{SKey(B)} \\ \text{Message 3.} & B \rightarrow S & : & \{i_b\}_{SKey(B)} \\ \text{Message 4.} & S \rightarrow A & : & \{n_s,\ i_a,\ i_b\}_{SKey(A)} \end{array}$$

where i_a and i_b are index values freshly introduced by A and B respectively, and n_s is a nonce freshly introduced by S. $SKey(A)$ is a previously established symmetric key shared only between A and S, assumed to be secure.

If S is modelled as internal, then its functionality is captured by all instantiations of the generation set G_2:

$$\langle n_s \rangle,\ \{B,\ i_a\}_{SKey(A)} \ \vdash\ \{n_s,\ \{A,\ i_a\}_{SKey(B)}\}$$

for the creation of message 2's with a fresh value bound to n_s, and all instantiations of the deduction set D_4:

$$\left\{ \begin{array}{l} \{B,\ i_a\}_{SKey(A)}, \\ n_s,\ \{A,\ i_a\}_{SKey(B)}, \\ \{i_b\}_{SKey(B)} \end{array} \right\} \ \vdash\ \{n_s,\ i_a,\ i_b\}_{SKey(A)}$$

for the creation of message 4's.

If we allow the intruder to perform deductions in D_4 whenever he likes, there is nothing stopping him from binding any nonce value to n_s, even if it does not correspond to the one introduced by S in the preceding message 2. Consider the following sequence of events, where Alice and Bob take the roles of A and B respectively:

(1) *Message 1.* $Alice \ \rightarrow\ I_{Server}\ :\ \{Bob,\ I_A\}_{SKey(Alice)}$ *(I_A fresh)*

(2) *(Generation) The intruder generates message 2 on behalf of S:*

$$\langle N_S \rangle,\ \{\{Bob,\ I_A\}_{SKey(Alice)}\} \ \vdash\ \{N_S,\ \{Alice,\ I_A\}_{SKey(Bob)}\}$$

(3) *Message* 2. $I_{Server} \rightarrow Bob : N_S, \{Alice, I_A\}_{SKey(Bob)}$

(4) *Message* 3. $Bob \rightarrow I_{Server} : \{I_B\}_{SKey(Bob)}$

(5) *(Deduction) The intruder deduces the following message* 4*, supposedly on behalf of S:*

$$\left\{ \begin{array}{l} \{Bob,\ I_A\}_{SKey(Alice)}, \\ \{I_B\}_{SKey(Bob)}, \\ N_I,\ \{Alice,\ I_A\}_{SKey(Bob)} \end{array} \right\} \vdash \{N_I,\ I_A,\ I_B\}_{SKey(Alice)}$$

where $N_I \neq N_S$ *and introduced by the intruder.*

(6) *Message* 4. $I_{Server} \rightarrow Alice : \{N_I, I_A, I_B\}_{SKey(Alice)}$

■

On receiving message 4 in step (6), *Alice* believes that N_I is a fresh nonce that the server introduces in this run. Given the assumption that the encryption $SKey(Alice)$ is *secure* (known only by *Alice* and the server) and that it contains the *Alice's* fresh index value I_A, this message 4 could only have been generated by the server. However, the deduction performed in step 5 does not accurately reflect the functionality of S, since S only constructs message 4's containing values bound to n_s that are the same as the corresponding values S introduces as fresh in message 2 (in this example, that would be N_S and not N_I).

This type of *false* behaviour by the intruder is an artifact of our model through the internalisation of the server role S, rather than a true reflection of the intruder's abilities. If S was modelled as a standard external agent, then the intruder would not be able to *deduce* the message 2 in step 5 above (since he would not have the deduction set of S to misuse). Although trivial, this example does illustrate how false behaviour (potentially leading to false attacks) could arise as a result of modelling agents internally within the intruder.

It is worth noting at this point that if message 2 was defined as follows:

Message 2. $S \rightarrow B : \{n_s, A, i_a\}_{SKey(B)}$

then the intruder would not be able to take advantage of the server's deductions, since the fresh value introduced in message 2 is bound under the secure encryption in the deduction set for message 4, namely:

$$\left\{ \begin{array}{l} \{B,\ i_a\}_{SKey(A)}, \\ \{n_s,\ A,\ i_a\}_{SKey(B)}, \\ \{i_b\}_{SKey(B)} \end{array} \right\} \ \vdash \ \{n_s,\ i_a,\ i_b\}_{SKey(A)}$$

In order to understand why the internalisation of agents can introduce false attacks, we need to address the following question: Given an agent A, what properties are satisfied when A is modelled as an external agent, but are subsequently lost when we model A as internal? Once we have identified these characteristics, we can reduce the likelihood for false attacks of this nature by carefully imposing constraints upon the intruder that prevents him from misusing the internal agent's deductions in this manner. Looking back at Example 5.4, we would expect such conditions to forbid the intruder from performing the deduction of S in step 5.

5.2.1 External agents versus internal agents

Our aim here is to identify the key factors that cause false attacks to arise when agents are internalised, as opposed to them being modelled as external processes. The observation we make is that by internalising an agent A, we lose the notion of A being able to distinguish between runs through the fresh values A introduces and being able to determine how these values are used in subsequent messages sent by A. This occurs because A no longer is bound by the external process structure and ordering of messages, as defined by the protocol description. The extent to which the intruder can take advantage of this is entirely dependent on the protocol description (as discussed in the previous section).

If a trustworthy agent A is modelled as external, then the fact that A has a *memory* spanning over the lifetime of each run he participates in, means that A is able to ensure the *freshness* of the runs with regards to the values he introduces freshly in each case. If A introduces a fresh value v in some message M (bound to a variable x), then all instances of the variable x in any subsequent messages sent by A will be bound to v only (for the duration of that run). In the case where protocols rely on the freshness and uniqueness of values by one or more of their trustworthy agents for each run, it is necessary that these agents have the memory capacity during a run to verify that any messages received and sent contain the correct fresh values bound to the correct variables expected.

Using Example 5.4 to clarify this, consider the case where the server S is modelled as an external process. For every message 2 that S generates with a fresh value v bound to the variable n_s, S will only ever generate one message 4 where n_s is assigned the value v in any given run. In addition, S knows the values bound to the other variables in the message 2 he sent (where v is introduced) and will ensure that these bindings are maintained in message 4. Hence, given the first 4 steps performed in this example, the only message 4 that S would be prepared to send (as an external process) in step 5 is the following:

$$\text{Message 4.} \quad Sam \rightarrow I_{Bob} \ : \ \{N_S,\ I_A,\ I_B\}_{SKey(Alice)}$$

In essence, for a given external agent A, if we observe the messages created and sent by A only, then there is a *control flow* determined solely by A, for each fresh value introduced by A. By control flow we mean that for every run R, A knows each precise fresh value v he introduces and binds to a given variable x, and for all subsequent messages that A sends throughout R, all instances of x will be assigned the same fresh value v. Furthermore, there is a fixed ordering of the occurrence of each fresh value in the protocol messages introduced and sent by A, and corresponds to the ordering of the messages in the protocol description. For example, if A introduces a fresh value v in a message 2 and subsequently sends messages 4 and 6 containing v, then we know that A will send the message 4 first, followed by the message 6.

Thus, in order to reduce the likelihood of false attacks, we need to carefully restrict the intruder in such a way that we regain the notion of control flow with regards to the fresh values introduced by internal agents. We achieve this by introducing special constraining processes into our models, referred to as *Supervisor processes* and presented below.

5.2.2 Supervisor processes

As we mentioned earlier in this chapter, most of our protocol models do not suffer from false attacks. Occasionally though, this problem can arise due to the loss of properties discussed above when an agent is modelled as internal. In these cases, we implement special constraint processes, referred to as *Supervisor processes*, into the offending models to reduce their likelihood. In this section, we give a brief overview of their design.

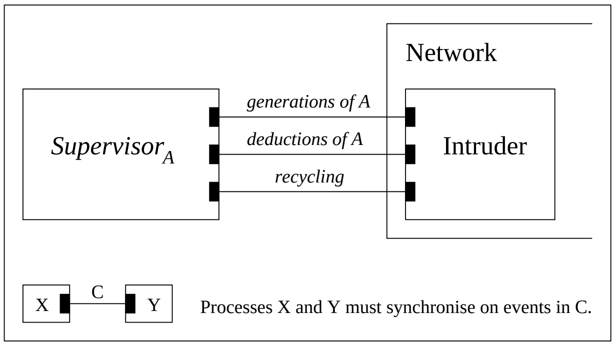

Figure 5.1: Supervisor process for internal agent A.

For a given internal agent A, a *supervisor process* is responsible for reflecting the fact that A is able to control and verify the use of the fresh values A introduces throughout a protocol run. No restrictions must be placed upon the actual introduction of a fresh value v, since that is the sole responsibility of the corresponding manager process. Instead, once v is introduced, the supervisor process is expected to control how v is used in subsequent messages generated on behalf of A, together with the ordering these messages are generated in (according to the protocol description).

Using Example 5.4 as an example, for every message 2 the intruder generates on behalf of the server S, we expect the supervisor process to allow the intruder to deduce precisely *one* message 4, where all the values (including the fresh nonce introduced by S) correspond to those in message 2. If no such message 2 is generated, then the supervisor should prevent the intruder from deducing any *false* message 4 (through deductions of S). Thus, the role of the supervisor process is to prevent the intruder from deducing any message 4's on behalf of S that were not preceded by the generation of a corresponding message 2. Note that the intruder is free to use his own deductive system to deduce any messages he likes. The supervisor process only deals with the deductions and generations of the internal agents.

Thus, for a given internal agent A, a supervisor process essentially introduces an abstract notion of a memory for A, in the sense that it ensures that messages generated by A obey the ordering they normally would have, had A been defined as an external process (as defined by the protocol description). Furthermore, it reflects the fact that A can verify and recognise its own fresh values introduced for the duration of a protocol run.

A consequence of imposing constraints upon the deductions and generations of internal agents, is that the supervisor processes must also take the recycling of fresh values into account. If an internal agent A introduces a fresh value v in some run R, then it is possible that v might get recycled (all external agents that know v have completed or withdrawn from the run). However, this does not guarantee that A has finished or withdrawn from the run R. It is plausible that A is still willing and able to send further messages containing v (in the sense that there may still be deductions and generations involving v that the intruder can perform on behalf of A). Once recycled, these events can no longer take place with v. The supervisor process for A must take this into account. Using Example 5.4 once again as an example, consider the following trace of events:

(1) Message 1. $Alice \rightarrow I_{Server} : \{Bob,\ I_A\}_{SKey(Alice)}$

(2) *(Generation)* The intruder generates message 2 on behalf of S:

$$\langle N_S \rangle,\ \{\{Bob,\ I_A\}_{SKey(Alice)}\} \ \vdash\ \{N_S,\ \{Alice,\ I_A\}_{SKey(Bob)}\}$$

(3) Message 2. $I_{Server} \rightarrow Bob : N_S,\ \{Alice,\ I_A\}_{SKey(Bob)}$

(4) Message 3. $Bob \rightarrow I_{Server} : \{I_B\}_{SKey(Bob)}$

(5) *Alice* and *Bob* withdraw from the protocol run.

(6) N_S gets recycled to the stale value N_{old}.

If modelled as an external process, S would still be prepared to send a corresponding message 4 (although in this case, N_S would not have been recycled). To reflect this in the case where S is modelled as internal, the supervisor process should allow precisely one deduction of message 4 to take place with the recycled value, namely:

$$\left\{ \begin{array}{l} \{Bob,\ I_A\}_{SKey(Alice)}, \\ \{I_B\}_{SKey(Bob)}, \\ N_{old},\ \{Alice,\ I_A\}_{SKey(Bob)} \end{array} \right\} \ \vdash\ \{N_{old},\ I_A,\ I_B\}_{SKey(Alice)}$$

Forbidding such deductions or generations to take place under these circumstances would be too restrictive upon the internal agents. In turn, this could prevent the intruder from gaining access to information (potentially beneficial towards attacking the protocol) that he otherwise would have, had they been modelled as an external processes. We must therefore ensure that the supervisor processes do not restrict any legitimate behaviour the intruder can perform on behalf of the internal agents.

Designing and implementing supervisor processes within the CSP protocol models is straightforward. They are implemented as constraint processes and placed in parallel with the intruder component, synchronising upon all relevant deductions and generations of internal agents, together with the recycling events of fresh values; this is illustrated in Figure 5.1. The supervisor process for each internal agent A simply reflects the ordering of messages of A as defined by the protocol description and by the representation of A when modelled as an external process, and imposes this structure upon the intruder. This extension does not contribute to any state space explosions, since no additional behaviour is enabled; if anything, there would be a slight decrease in state space because the potential for false behaviour by the intruder is reduced.

5.3 Extending CSP protocol models and Casper

In this section, we give an overview of the main extensions made to the CSP protocol models to handle internal agents. We then present the simple extension made to the Casper input script that prompts Casper to automatically use this approach in the models it generates, for the specified agents. For now, we will assume that we are only internalising agents that are not part of any specifications; Section 5.5 extends this work to consider the case where we need to construct signal events for internal agents by appropriate renamings of their corresponding deductions and generations.

5.3.1 Extending the CSP protocol models

There are two main components within our CSP protocol models that need to be extended to incorporate internal agents. The first of these is the intruder process; we discussed this extension in detail earlier in Section 5.1. The second (and more subtle) one is the design of the manager processes; we present an overview of this work here.

As discussed in Chapter 4, a manager process $Manager_T$ for some type T is responsible for supplying the network with fresh values of this type, and for each value v, monitoring the learning and forgetting of v by the agent processes throughout the run of the system. This is achieved by placing $Manager_T$ in parallel with the network (agents and intruder) and making them synchronise upon the following events: i) all *send* events that introduce one or more fresh values of type T, ii) all *receive* events that involve agents learning fresh values of type T for the first time, and iii) *close* events indicating the withdrawal or termination of agents, thereby forgetting any fresh values they may have stored. This last group of events enables the recycling of fresh values to be triggered in the case where they have been forgotten by all agent processes.

With regards to internal agent models, it is necessary to divide the manager processes into the following two types: *internal* managers and *external* managers. We say that a manager process for some type T is *internal* precisely when there exists an internal agent A that introduces fresh values of type T (in messages sent by the intruder on behalf of A). A manager process that is not *internal* is referred to as *external*. Thus, all manager processes presented in Chapter 4 (before we considered internal agents) were external. Consider the first two messages of the adapted NSPK protocol:

Message 1. $A \rightarrow B \ : \ \{n_a,\ A\}_{PK(B)}$
Message 2. $B \rightarrow A \ : \ \{n_a,\ n_b,\ B\}_{PK(A)}$

If either or both of the agents A and B were modelled as internal, then the nonce manager would be an internal one, because both A and B introduce fresh nonces bound to n_a in message 1's and n_b in message 2's respectively.

The design of an internal manager for some type T is somewhat more intricate than that of the corresponding external one, since it needs to synchronise with the intruder upon all the *generate* events that require fresh values of type T, in addition to the standard synchronisations summarised above. Recall that *generate* events represent the generations on behalf of internal agents. For example, in the adapted NSPK protocol, if the agent B is modelled as internal, then the nonce manager must synchronise with all the *generate.X* events where X takes the form:

$$\langle n_b \rangle,\ \{n_a,\ A\}_{PK(B)} \ \vdash \ \{n_a,\ n_b,\ B\}_{PK(A)} \tag{5.1}$$

and is responsible for supplying the fresh nonce values bound to n_b.

This extension in itself is quite straightforward. However, the following subtle problem arises as a direct result. As it stands, no restrictions are placed upon the intruder with regards to performing generations on behalf of the internal agents. If we simply allow the intruder to perform them whenever he likes, then he could perform an unbounded number of them, thus causing the manager processes to run out of fresh values. Using the generation in (5.1) above, he can do this, for example, by using the same message 1 in the antecedent to generate many different message 2's, each characterised by a different supply of fresh nonces bound to n_b. Furthermore, he can do this without necessarily passing them on to agent processes, or even without the actual need for them in the system; he can simply store the resulting messages in his memory.

To overcome this problem, we derive conditions upon the intruder restricting the number of generations he can perform and the number of fresh values he can store at any one time, unknown to any agent processes. Deriving these bounds and justifying them has proved to be one of the most troublesome problems in our work. Chapter 6 is devoted to addressing this issue and takes it further to show that, for a given class of protocols, we can derive bounds such that if an attack exists upon the protocol of unbounded size (i.e. without the boundaries and with an unbounded supply of fresh values), then an equivalent attack will also be found upon the corresponding finite protocol model with these bounds in place.

For now, we only consider how these conditions are implemented within the internal manager processes; justifying them will be addressed in Chapter 6. Thus, an internal manager for some type T will agree to synchronise upon the relevant *generate* events, provided that the number of fresh values of type T the intruder is currently storing, unknown to any agent processes, does not exceed some bound Max_T. Intuitively, implementing these conditions is straightforward, since it is a case of calculating the set difference between the set of fresh values known to the intruder and the set of those known by agent processes.

In practice, constructing these sets are slightly more complicated, due to the highly parallelised design of the manager processes. Recalling from Chapter 4, a manager process for some type T is composed of the parallel composition of sub-processes, one for each fresh value of this type. A sub-process for a fresh value v, is only responsible for monitoring the agent processes as they learn and forget v throughout the run of the system. Thus, it is easy to compute the set of agents that store v at any given point in time. However, there is no global memory within the manager processes that directly answers the question: For a given agent process A, what is the complete set of fresh values known by A at a given point in time?

Initially, we reverted back to the equivalent sequential design of the manager processes, thereby making these computations very straightforward (as all the necessary information was stored globally). The disadvantage of this design is that it is very inefficient during the compilation stages in FDR; the growth is exponential with respect to the number of fresh values declared. Typically, we found that if the cardinality of these sets were above 3, the compilation phase would start to become unmanageably large (depending also upon the complexity of the protocol being modelled).

Thus, the most efficient approach is to maintain the highly parallel structure and add some careful synchronisations between the components (these remain internal within the manager processes) that provide the necessary information to construct the conditions upon the *generate* events required. The problem of compilation time is removed as the growth is linear with respect to the number of fresh values declared. This is indeed how our internal manager processes are implemented.

5.3.2 Extending Casper

The process of implementing agents using our internal representation within the CSP protocols models has been fully automated within Casper. As is the case with all the developments presented throughout this thesis that have been automated, this abstracts away the complexity of the models and the level of knowledge required to implement them, making them accessible to a broader class of users.

In the Casper input script, the extension made is very minor. Users simply specify the agent roles to be internalised within the `#Intruder Information` by adding the following line:

$$\texttt{IntruderProcesses = A}_1\texttt{,...,A}_n$$

where $\texttt{A}_1\texttt{,...,A}_n$ are the agent roles to be internalised. These roles should not also be instantiated in the actual system definition (in the `#System`). At this point, Casper assumes that a process is either instantiated in the system or modelled as internal. In the latter case, the process should not be part of the specifications. Casper generates appropriate error messages if these requirements are violated.

As we will illustrate in some examples in the next section, the behaviour of the internal agents are represented by the corresponding *generate* and *deduce* events; in particular, such an event represents the construction of a new message by an internal agent. In the case where such events contribute towards an attack upon the protocol, they will be present in the trace of a corresponding counter-example found by FDR. Users of Casper simply pass these results back through Casper using the *interpret* function to get a higher level description of the attack; therefore, we extend this function to give a more descriptive presentation for the events performed on behalf of the internal agents. For example, the event:

```
generate.RESPONDER.(<Nb>,{Encrypt.(PK__.Bob,<Na, Alice>)},
                         {Encrypt.(PK__.Alice,<Na, Nb, Bob>)})
```

is interpreted as:

"The intruder performs the functionality of the internal process RESPONDER and generates the set of messages: $\{\{Na,\ Nb,\ Bob\}_{PK(Alice)}\}$ *from the set of messages (he knows):* $\{\{Na,\ Alice\}_{PK(Bob)}\}$*, and the fresh value(s) Nb supplied."*

using the *linterpret* function for LaTeX presentations. This provides some useful intuition towards understanding attacks that involve internal agents. We advise Casper users to model server-type roles (not involved in the specifications being verified) as internal, since these models are much more efficient in terms of state space in FDR. We illustrate this in the next section with some examples.

5.4 Some examples: Internal servers

In this section, we apply our new modelling approach to two protocol examples, the Otway-Rees and the Yahalom protocol, and demonstrate the impact it has upon the resulting state space.

5.4.1 Example 1: The Otway-Rees protocol

The version we are considering is the same as the one we considered in Chapter 4 (based on [BAN89]):

Message 1. $A \rightarrow B \ : \ n_a$
Message 2. $A \rightarrow S \ : \ \{n_a,\ A,\ B\}_{SKey(A)}$
Message 3. $B \rightarrow S \ : \ n_a,\ n_b$
Message 4. $B \rightarrow S \ : \ \{n_a,\ A,\ B\}_{SKey(B)}$
Message 5. $S \rightarrow B \ : \ n_a,\ \{n_b,\ k_{ab}\}_{SKey(B)}$
Message 6. $S \rightarrow A \ : \ \{n_a,\ k_{ab}\}_{SKey(A)}$

This will allow us to make some useful comparisons with regards to the effects that internalising the server has upon the state space.

System verified

We verify the same system as we did previously in Chapter 4, except that this time we model the server role S as internal. Thus, in the Casper input script, the actual system declared in `#System` consists of the following two agents:

```
Initiator(Alice, Sam)
Responder(Bob, Sam)
```

where `Initiator` represents the role A and `Responder` represents the role B. The server role is modelled as internal by adding the following line in the `#Intruder Information`:

```
IntruderProcesses = Server
```

We declare the types *Nonce* (types of n_a and n_b) and *SessionKey* (type of k_{ab}) as data independent; the manager processes will be responsible for supplying fresh values for these variables when required. We declare the set of available fresh nonces and keys as:

$$\begin{array}{rcl} Nonce_F & = & \{N_1,\ N_2\} \\ SessionKey_F & = & \{K_1,\ K_2\} \end{array}$$

and the known and unknown background (stale) nonces and keys as:

$$\begin{array}{rcl} KnownBackground\,(Nonce) & = & NB_p \\ KnownBackground\,(SessionKey) & = & KB_p \\ UnknownBackground\,(Nonce) & = & NB_s \\ UnknownBackground\,(SessionKey) & = & KB_s \end{array}$$

Once again, we are testing our system for the following `Casper` authentication specification: `Agreement(B, A, [])`, specifying that if A thinks she has successfully completed a run of the protocol with B, then B has previously been running the protocol, apparently with A, under the following conditions: i) Both agents agreed as to which roles they took, and ii) there is a one-to-one relationship between the runs of A and those of B.

Resulting analysis

Without the use of stale knowledge, the following authentication attack was found:

(1) Message 1. $Alice \ \rightarrow\ I_{Alice}\ :\ N_1$

(2) Message 2. $Alice \ \rightarrow\ I_{Sam}\ :\ \{N_1, Alice, Alice\}_{SKey(Alice)}$

(3) *(Generation 1)* The intruder performs the functionality of the server *Sam* to generate messages 5 and 6, by the following generation:

$$\langle K_1 \rangle,\ \left\{ \begin{array}{l} \{N_1, Alice, Alice\}_{SKey(Alice)}, \\ N_1,\ NBp \end{array} \right\} \vdash \left\{ \begin{array}{l} \{N_1, K_1\}_{SKey(Alice)}, \\ \{NB_p, K_1\}_{SKey(Alice)} \end{array} \right\}$$

If *Sam* was modelled as external, then this would be captured by:

$$\begin{array}{lrcl}
\text{Message 2.} & I_{Alice} \rightarrow Sam & : & \{N_1,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 3.} & I_{Alice} \rightarrow Sam & : & N_1,\ NB_p \\
\text{Message 4.} & I_{Alice} \rightarrow Sam & : & \{N_1,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 5.} & Sam \rightarrow I_{Alice} & : & N_1,\ \{NB_p,\ K_1\}_{SKey(Alice)} \\
\text{Message 6.} & Sam \rightarrow I_{Alice} & : & \{N_1,\ K_1\}_{SKey(Alice)}
\end{array}$$

(4) Message 6. $I_{Sam} \rightarrow Alice : \{N_1,\ K_1\}_{SKey(Alice)}$

(5) *Alice* withdraws from this run as *Initiator*.

(6) K_1 has been recycled to KB_s and is therefore regarded as fresh again.

(7) N_1 has been recycled to NB_p and is therefore regarded as fresh again.

(8) Message 1. $Alice \rightarrow I_{Bob} : N_1$

(9) Message 2. $Alice \rightarrow I_{Sam} : \{N_1,\ Alice,\ Bob\}_{SKey(Alice)}$

(10) *(Generation 2)* The intruder performs the functionality of the server *Sam* to generate messages 5 and 6 by the following generation:

$$\langle K_1\rangle,\ \left\{\begin{array}{l} \{NB_p, Alice, Alice\}_{SKey(Alice)} \\ NB_p,\ N_1 \end{array}\right\} \vdash \left\{\begin{array}{l} \{N_1, K_1\}_{SKey(Alice)}, \\ \{NB_p, K_1\}_{SKey(Alice)} \end{array}\right\}$$

If *Sam* was modelled as external, then this would be captured by:

$$\begin{array}{lrcl}
\text{Message 2.} & I_{Alice} \rightarrow Sam & : & \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 3.} & I_{Alice} \rightarrow Sam & : & NB_p,\ N_1 \\
\text{Message 4.} & I_{Alice} \rightarrow Sam & : & \{NB_p,\ Alice,\ Alice\}_{SKey(Alice)} \\
\text{Message 5.} & Sam \rightarrow I_{Alice} & : & NB_p,\ \{N_1,\ K_1\}_{SKey(Alice)} \\
\text{Message 6.} & Sam \rightarrow I_{Alice} & : & \{NB_p,\ K_1\}_{SKey(Alice)}
\end{array}$$

(11) Message 6. $I_{Sam} \rightarrow Alice : \{N_1, K_1\}_{SKey(Alice)}$

On receiving the final message 6, *Alice* believes she has completed a run of the protocol (as the *Initiator*), with *Bob* (as the *Responder*). The truth is that *Bob* was never actually running the protocol; the intruder successfully impersonated him.

This (well-known) attack is identical to the one we presented in our example of this analysis in Chapter 4. The only difference is the way in which the functionality of the *Server* role is captured in steps (3) and (10) by *Generation 1* and *Generation 2* respectively. Once again, a previous run of the system was required in order to give the intruder the stale messages he needs to break the protocol and construct his attack.

The number of states required to find this attack was 25,556 states; this is certainly an improvement upon the 801,919 states required to find this attack for the same system where the server was modelled as an external process (without the use of stale knowledge). However, we can improve the efficiency of the search further by adding the stale knowledge option to the intruder. The equivalent attack is then found in 168 states and is represented as follows:

(1) Message 1. $Alice \ \rightarrow \ I_{Bob} \ : \ N_1$

(2) Message 2. $Alice \ \rightarrow \ I_{Sam} \ : \ \{N_1, Alice, Bob\}_{SKey(Alice)}$

(3) *(Generation 1)* The intruder performs the functionality of the server *Sam* to generate messages 5 and 6, by the following generation:

$$\langle K_1 \rangle, \left\{ \begin{array}{l} \{NB_p, Alice, Alice\}_{SKey(Alice)}, \\ \{NB_p, Alice, Alice\}_{SKey(Alice)}, \\ NB_p, \ N_1 \end{array} \right\} \vdash \left\{ \begin{array}{l} \{NBp, K_1\}_{SKey(Alice)}, \\ \{N_1, K_1\}_{SKey(Alice)} \end{array} \right\}$$

If *Sam* was modelled as external, then this would be captured by:

$$\begin{array}{lrcl} \text{Message 2.} & I_{Alice} \rightarrow Sam & : & \{NB_p, \ Alice, \ Alice\}_{SKey(Alice)} \\ \text{Message 3.} & I_{Alice} \rightarrow Sam & : & NB_p, \ N_1 \\ \text{Message 4.} & I_{Alice} \rightarrow Sam & : & \{NB_p, \ Alice, \ Alice\}_{SKey(Alice)} \\ \text{Message 5.} & Sam \rightarrow I_{Alice} & : & NB_p, \ \{N_1, \ K_1\}_{SKey(Alice)} \\ \text{Message 6.} & Sam \rightarrow I_{Alice} & : & \{NB_p, \ K_1\}_{SKey(Alice)} \end{array}$$

The intruder makes use of stale messages in this series of replays.

(4) Message 6. $I_{Sam} \ \rightarrow \ Alice \ : \ \{N_1, K_1\}_{SKey(Alice)}$

As discussed in more detail in Chapter 4, using the stale knowledge option means that the intruder is given the stale messages from all possible previous runs. Hence, our search no longer needs to explore these states, since they are part of the initial knowledge of the intruder.

To summarise, the following table illustrates the effects our state space reduction strategies have upon the state space:

	Server external		**Server internal**	
Stale knowledge	False	True	False	True
States (for attack)	801,919	7,747	25,556	168

5.4.2 Example 2: The Yahalom protocol

As a second example, consider the Yahalom protocol defined by the following sequence of messages:

$$\begin{array}{lll}
\text{Message 1.} & A \to B & : \; n_a \\
\text{Message 2.} & B \to S & : \; n_b,\ \{A,\ n_a\}_{SKey(B)} \\
\text{Message 3.} & S \to A & : \; n_b,\ \{B,\ k_{ab},\ n_a\}_{SKey(A)} \\
\text{Message 4.} & S \to B & : \; n_b,\ \{A,\ k_{ab},\ n_b\}_{SKey(B)} \\
\text{Message 5.} & A \to B & : \; \{n_b\}_{k_{ab}}
\end{array}$$

where n_a and n_b are nonces freshly introduced by the agents A and B, and k_{ab} is a session key freshly introduced by the server S. This version is closely based on the one suggested in [BAN89]. As was the case with the Otway-Rees protocol example, the only difference is the way in which the server communicates message 4 directly to B rather than using A as a messenger; this type of re-direction has no effect on security (Hui and Lowe [HL01]).

This particular version of the Yahalom protocol has a well known authentication attack found by Syverson [Syv94] and illustrated in Chapter 2. For the corresponding CSP model capturing this protocol using the data independence techniques, the following table illustrates the effects of our state space reduction strategies upon the state space for finding this attack:

	Server external		**Server internal**	
Stale knowledge	False	True	False	True
States (for attack)	222,104	1,024	12,977	249

5.5 Signal events for internal agents

When an agent A is modelled as a standard external process, we capture the *state of mind* of A by analysing the messages *sent* and *received* by A. These events are then renamed to the *signal* events required for the specifications being verified. On the other hand, an internal agent no longer performs *send* and *receive* events, since its functionality is solely captured through deduction and generation sets within the intruder. It is for this reason we initially only considered internalising server-type roles that were not part of any of the specifications. In this section, we attempt to extend our models to allow agents to both be part of a specification and be modelled as internal. The question we need to address is: How do we construct signal events for an internal agent from its corresponding deduction and generation sets?

5.5.1 Constructing signal events for internal agents

To construct signal events for an internal agent A, we need to decide how to capture the notion of A sending and receiving messages. It turns out to be quite straightforward when it comes to capturing the sending of a message M by A, since this essentially corresponds to the deduction or generation of A resulting in M. All deductions and generations are performed as *deduce* and *generate* events within the intruder, so it is simply a case of identifying the ones of interest and renaming them to the corresponding signal events (as done previously). As an example, consider the first message of the NSPK protocol:

Message 1. $A \to B \ : \ \{n_a,\ A\}_{PK(B)}$

and suppose A is modelled as internal. The *creation* of message 1 is represented by the following generation:

$$\langle n_a \rangle,\ \{A,\ B\} \ \ \vdash \ \ \{n_a,\ A\}_{PK(B)}$$

where the value bound to n_a is freshly supplied by the Nonce Manager. If such a generation was performed, for example,

$$\langle N_A \rangle,\ \{Alice,\ Bob\} \ \ \vdash \ \ \{N_A,\ Alice\}_{PK(Bob)}$$

where *Alice* takes the role of A, introducing the fresh nonce N_A, then this is essentially the same as an externally modelled version that sends this message 1 to *Bob*, but is intercepted by the intruder. The system is in the same position in the sense that (in both cases) the intruder can chose whether to forward this message 1 on to *Bob* on behalf of *Alice* or not. This generation is performed upon the channel *generate*. Thus, if we need to construct a signal event bound to the sending of this message (by *Alice*), we simply rename it accordingly. For example,

$$generate\,.\,(\langle N_A\rangle,\ \{Alice,\ Bob\},\ \{N_A,\ Alice\}_{PK(Bob)})$$

is renamed to:

$$signal\,.\,Alice\,.\,Bob\,.\,N_A$$

As a second example, consider the final message of the NSPK protocol:

$$\text{Message 3.}\quad A \rightarrow B\ :\ \{n_b\}_{PK(B)}$$

and once again suppose A is modelled as an internal agent. The creation of this message is represented by the following deduction:

$$\{n_a,\ A\}_{PK(B)},\ \{n_a,\ n_b\}_{PK(A)}\ \ \vdash\ \ \{n_b\}_{PK(B)}$$

The signal events that represent the running of the protocol by A (used in the authentication specification of A to B) are bound to the sending of this message 3 by A. Thus, we can construct them through the renaming of:

$$deduce\,.\,(\{n_b\}_{PK(B)},\ \{\{n_a,\ A\}_{PK(B)},\ \{n_a,\ n_b\}_{PK(A)}\})$$

to the events:

$$Signal\,.\,Running\,.\,A\_role\,.\,A\,.\,B\,.\,n_a\,.\,n_b$$

On the other hand, constructing signal events for an internal agent A that are bound to the *receiving* of some message M by A is more complicated than the former case. Using the adapted NSPK protocol as an example, consider the case where the responder agent B is modelled as internal. B's functionality is captured by a single generation set for the creation of message 2's:

$$\langle n_b\rangle,\ \{n_a,\ A\}_{PK(B)}\ \ \vdash\ \ \{n_a,\ n_b,\ B\}_{PK(A)}$$

where the nonce bound to n_b is freshly supplied by the Nonce Manager.

Now suppose we are interested in the signal event representing the fact that B believes he has completed a run with A. By definition, this signal (commit) event is bound to the last message B participates in; in this case, message 3. B participates in message 3 as the *receiver*. So the question arises: How do we capture the act of the internal B *receiving* message 3, thereby allowing us to deduce his state of mind at that given point? The only generation we have (with respect to B) is the generation for message 2; this says nothing concerning the receipt of message 3 by B.

One way to address this problem is to introduce a special deduction, whose sole purpose is to simply *echo* all message 3's that B receives. The antecedent would be all the previous messages involving B, together with the receipt of message 3's. Essentially, we are converting the act of *receiving* of some message M by an internal agent A into the *sending* of M by A; we can then apply the simple renaming strategy above. Going back to our example of constructing signal events that represent the commit of B, we can now construct the following *echo* deduction set for B:

$$\left\{ \begin{array}{ll} (Msg\ 1) & \{A, n_a\}_{PK(B)}, \\ (Msg\ 2) & \{n_a,\ n_b,\ B\}_{PK(A)}, \\ (Msg\ 3) & \{n_b\}_{PK(B)} \end{array} \right\} \ \vdash \ \{n_b\}_{PK(B)}$$

and perform the appropriate renamings to:

$$Signal\ .\ Commit\ .\ B\_role\ .\ B\ .\ A\ .\ n_a\ .\ n_b$$

What we are essentially doing is extending the protocol description with an extra message as follows:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B & : \{A, n_a\}_{PK(B)} \\ \text{Message 2.} & B \rightarrow A & : \{n_a,\ n_b,\ B\}_{PK(A)} \\ \text{Message 3.} & A \rightarrow B & : \{n_b\}_{PK(B)} \\ \text{Message 3}(echo). & B \rightarrow Env & : \{n_b\}_{PK(B)} \end{array}$$

where Env represents the environment.

5.5.2 A basic requirement in practice

When constructing signal events from deductions or generations in practice, it becomes apparent that there is a basic requirement that must be satisfied by the messages involved in order to successfully achieve this.

Consider a signal event $Signal_A$ of an internal agent A with respect to some other agent B and variable set X (for example, one of the form $Signal.Running.A.B.X$). $Signal_A$ is constructed through the renaming of the corresponding deduction or generation C_A of A. In order to construct $Signal_A$, the identities of A and B, and the variables in X must be present in either the messages in the precondition of C_A or in the resulting message generated by C_A. Without this property, the deduction or generation corresponding to a signal event of an agent A with respect to another agent B, would not provide sufficient information regarding their identities and the values they are expected to agree on. This property is not restrictive to the class of protocols we can consider, since it is satisfied by most protocols.

5.6 Example: Specifications with internal agents

In this example, we demonstrate how we construct signal events for internal agents in practice. To keep this example simple and intuitive, we consider the original version of the Needham-Schroeder Public-Key (NSPK) protocol [NS78] defined as follows:

Message 1. $A \rightarrow B \ : \ \{A, n_a\}_{PK(B)}$
Message 2. $B \rightarrow A \ : \ \{n_a,\ n_b\}_{PK(A)}$
Message 3. $A \rightarrow B \ : \ \{n_b\}_{PK(B)}$

The authentication attack found by Lowe [Low96a] is as follows:

Message 1.(α) $Alice \rightarrow I_{Mallory} \ : \ \{N_1,\ Alice\}_{PK(Mallory)}$
Message 1.(β) $I_{Alice} \rightarrow Bob \ : \ \{N_1,\ Alice\}_{PK(Bob)}$
Message 2.(β) $Bob \rightarrow I_{Alice} \ : \ \{N_1,\ N_2\}_{PK(Alice)}$
Message 2.(α) $I_{Mallory} \rightarrow Alice \ : \ \{N_1,\ N_2\}_{PK(Alice)}$
Message 3.(α) $Alice \rightarrow I_{Mallory} \ : \ \{N_2\}_{PK(Mallory)}$
Message 3.(β) $I_{Alice} \rightarrow Bob \ : \ \{N_2\}_{PK(Bob)}$

This is an attack upon the authentication of *Alice* to *Bob* since, *Bob* believes he has completed a run of the protocol (taking role B) with *Alice*, using data items N_1 and N_2. However, *Alice* believes she is running the protocol (taking role A) with *Mallory*, using data items N_1 and N_2.

Using data independence and internal agents

We now present the CSP model corresponding to the NSPK protocol, where we make full use of the data independence techniques, together with the work presented in Section 5.5 concerning the construction of signal events for internal agents. The CSP model is designed as follows. We model the initiator agent A as internal within the intruder, together with one instance of B, with identity *Bob*, as an external agent. We are interested in verifying the authentication of A to B, and therefore require the following two classes of signal events:

Signal.Running events for all instances of A running internally. These are constructed through the renaming of all message 3's deduced on behalf of A by the intruder.

Signal.Commit events for *Bob* playing the role of B. These are captured through the standard renaming procedure of his *receive* events.

Capturing the authentication attack

Lowe's authentication attack presented above, is captured as follows:

(1) The intruder generates message 1 on behalf *Alice* as role A:

$$\{\langle N_1 \rangle,\ \{Alice,\ Mallory\}\} \quad \vdash \quad \{N_1,\ Alice\}_{PK(Mallory)}$$

If A was modelled as external, then this would be captured by:

$$\text{Message 1.} \quad Alice \rightarrow I_{Mallory}\ :\ \{N_1,\ Alice\}_{PK(Mallory)}$$

(2) Message 1. $I_{Alice} \ \rightarrow\ Bob\ :\ \{N_1,\ Alice\}_{PK(Bob)}$

(3) Message 2. $Bob\ \rightarrow\ I_{Alice}\ :\ \{N_1,\ N_2\}_{PK(Alice)}$

(4) The intruder deduces message 3 on behalf of *Alice* (as role *A*):

$$\left\{ \begin{array}{l} \{N_1,\ Alice\}_{PK(Mallory)}, \\ \{N_1,\ N_2\}_{PK(Alice)} \end{array} \right\} \vdash \{N_2\}_{PK(Mallory)}$$

If *A* was modelled as external, then this would be captured by:

$$\text{Message 3.} \quad Alice \rightarrow I_{Mallory} \ : \ \{N_2\}_{PK(Mallory)}$$

(5) Message 2. $I_{Alice} \ \rightarrow \ Bob \ : \ \{N_2\}_{PK(Bob)}$

The *Signal.Running* event for *Alice* in this attack was constructed by renaming the *Deduction* in Step 4 to the corresponding signal event:

$$deduce\,.\,(\{N_2\}_{PK(Mallory)},\ \{\{N_1,\ Alice\}_{PK(Mallory)},\ \{N_1,\ N_2\}_{PK(Alice)}\})$$

represents *Alice* constructing message 3 (as *A*) and is therefore renamed to:

$$Signal\ .\ Running\ .\ A\_role\ .\ Alice\ .\ Mallory\ .\ N_1\ .\ N_2$$

The *Signal . Commit* event for *Bob* with respect to his *perceived* run with *Alice* was constructed in the standard way through the following renaming of *Bob*'s event representing the act of him receiving message 3 as follows:

$$receive\ .\ Alice\ .\ Bob\ .\ (Msg3,\ \{N_2\}_{PK(Bob)})$$ [1]

is renamed to:

$$Signal\ .\ Commit\ .\ B\_role\ .\ Bob\ .\ Alice\ .\ N_1\ .\ N_2$$

In the CSP trace error representing the attack, the *Signal . Commit* event of *Bob* with respect to *Alice* occurs without the corresponding *Signal . Running* event of *Alice* with *Bob* (since *Alice* believed she was running with *Mallory*) preceding it.

[1] Simplified for the purposes of this example.

5.7 Discussion

In this chapter, we presented an alternative approach to modelling agent processes, known as *internal* agents. This involves capturing their functionality as a series of special deductions within the intruder component, instead of modelling them as individual CSP processes. This modelling technique has proved to be extremely advantageous to our model checking methods in two important ways.

The first contribution is as an effective state space reduction strategy. This was clearly demonstrated by the two protocol examples we presented in Section 5.4. The reduction of state space in both cases (and in all protocol models where this technique has been applied) proved significant (by an order of magnitude); this effect is even more substantial when no attack exists and the entire state space has to be explored. We also illustrated how the combination of this modelling technique and the use of the stale knowledge (state space reduction strategy introduced in Chapter 4) has the resulting effect of reducing the state space around 3 orders of magnitude. In the world of model checkers, these results are extremely useful, since one of the continuous battles is to control the rapid growth of states as the systems become larger. Developing these techniques, means that applying our methods to larger problems becomes more feasible than previously possible.

The second contribution made by the work presented in this chapter is to provide the foundations upon which the work in Chapter 6 is built. The notion of internal agents has provided a way in which we are able to capture higher degrees of parallelism than feasibly possible hitherto. In Chapter 6, we develop this idea and show that, for some classes of protocols, we are able to capture attacks for any degree of parallelism amongst the agent identities, whilst still performing finite refinement checks through FDR. The internal agent model has been automated within Casper, allowing a broader class of users to benefit from these results.

One problem we encountered when internalising agents, was the potential of false attacks. This proved to be uncommon in the cases where the roles internalised are of the server-like nature discussed above. False attacks have the potential of becoming more prominent in models where more general agent roles (for example, initiators) are internalised. In Section 5.2, we looked at this problem more carefully and proposed a solution for reducing their likelihood. This involved introducing special constraining processes into our CSP protocol models, known as *Supervisor processes*. Since these processes are usually not required, we currently implement them by hand into the models. However, it is straightforward to automate this process within Casper and is currently in the process of being accomplished.

Chapter 6

Capturing parallel attacks

In Chapter 5, we presented a new approach for modelling agents, known as *internal* agents. Instead of modelling an agent as a CSP process, its functionality is captured within the intruder component by a series of special deductions. Internal agents are divided into two categories. The first, and simpler one, contains those that generate messages without introducing any fresh values of some data independent type. Internal agents in this category are captured by deductions that take the same form as those used in the intruder's deductive system. An example is the server role in the TMN protocol (taken from [TMN90]) whose function is to receive two messages M_1 and M_3 and construct a corresponding third message M_4, where M_4 only contains variables in M_1 and M_3 (and so not introducing any fresh variables into the system). If we modelled this server role internally, then the corresponding deductions would be all valid instantiations of $(\{M_1, M_3\}, M_4)$.

The second class of internal agents are those that introduce fresh values. The functionality of such an agent is captured by a special type of deduction, known as a *generation*. For example, consider the following, which are the first two messages of a hypothetical protocol description:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow S & : \{B,\ n_a\}_{SKey(A)} \\ \text{Message 2.} & S \rightarrow A & : \{n_a,\ k_{ab}\}_{SKey(A)} \end{array}$$

Here A is an agent introducing the fresh nonce n_a and S is the server introducing the fresh key k_{ab}. If we modelled S as an internal server, then we would capture its functionality as the following generation:

$$\langle k_{ab}\rangle,\ \{B,\ n_a\}_{SKey(A)} \ \vdash\ \{n_a,\ k_{ab}\}_{SKey(A)} \tag{6.1}$$

As we discussed in Chapter 5, the corresponding key manager that is responsible for the supply of fresh keys will synchronise upon these generations and determine which fresh key is bound to k_{ab} each time such a generation takes place.

One of the problems that arises from this new modelling approach is that the intruder can perform any number of these generations he wishes, each time requesting a fresh key and thereby causing the key manager to run out of fresh values. The intruder can do this, for example, by using the same message 1 to generate many different message 2's, each characterised by a distinct fresh key. Furthermore, he can build up a store of these values and later use them one at a time with the honest agents. For essentially this reason, the recycling mechanism used elsewhere cannot necessarily be applied to these multiple message 2's held within the intruder. Using the *generation* of the internal server in (6.1) as an example, if the Key Manager is given a set of n fresh values to supply the network, and the intruder has intercepted the message 1 $\{Bob, N_A\}_{SKey(Alice)}$, then the intruder could legitimately perform the following sequence of generations:

$$\begin{array}{rcl} \langle K_1 \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_1\}_{SKey(Alice)} \\ \langle K_2 \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_2\}_{SKey(Alice)} \\ & \vdots & \\ \langle K_n \rangle,\ \{Bob, N_A\}_{SKey(Alice)} & \vdash & \{N_A, K_n\}_{SKey(Alice)} \end{array}$$

thereby always being able to cause the Key Manager to run out of values, irrespective of the value bound to n. The only way to keep the number of fresh values manageable (or even bounded) is to prevent the intruder storing many fresh values for later use.

Internal agents that generate fresh values create the most troublesome theoretical problem in our work: How can we reasonably limit the intruder's appetite for fresh values when it has the capability of requesting any number it wishes on behalf of the internal agents via the generations? In previous presentations of this work [RB99, BLR00], we derived bounds upon the intruder such that the resulting state space remained feasible for each protocol model. However, the danger of such bounds is that the intruder might be unreasonably restricted, resulting in attacks being lost. It may, for example, be advantageous for the intruder to store a number of messages, each containing fresh values and unknown to any external agents, so that he can rearrange them and use them later on in a run as part of an attack. In this chapter, we seek to address this issue and derive a number of conditions upon the intruder to limit his appetite for fresh values.

There are two main advantages for modelling agents internally within the intruder. The first is that this approach serves as an effective state space reduction technique, as discussed in Chapter 5. The second advantage, and one we will be focusing on in this chapter, is that the internal model of an agent A naturally captures a highly parallelised version of A. If A does not introduce any fresh values (for example, like the server in the TMN protocol, mentioned earlier), then the intruder is able to capture any degree of parallelism of A within the CSP model, by performing the standard deductions on behalf of A. On the other hand, if A does introduce fresh values, then the degree of parallelism of A that the intruder can capture is entirely dependent on the supply of fresh values. In a model where there is an infinite supply of them, the intruder is able to capture any degree of parallelism of A; however, if this supply is bounded, then the intruder may be restricted to only being able to perform a small number of instances of A in parallel at any one time.

In this chapter, we extend our CSP protocol models such that we are able to derive bounds upon the intruder's ability to request fresh values on behalf of internal agents, whilst at the same time enabling him to capture any degree of parallelism within the internal agents.

This chapter is arranged as follows. Section 6.1 introduces a new protocol model property, *just-in-time*, that we use to establish and justify finite bounds upon the intruder later on in this chapter. In Section 6.2, we present an extension to our CSP protocol models that involves introducing special values, referred to as *dummy* values. Together with the just-in-time property, this extension enables us to map a protocol model with an infinite supply of fresh values to an equivalent reduced system with a finite source of fresh values. Identifying the class of protocols that satisfy this property is not straightforward. Therefore we introduce a second useful property, referred to as *factorisability of internal agents* in Section 6.3. This property is used in Section 6.4 to establish three classes of protocols that satisfy the just-in-time property, together with the corresponding bounds derived upon the intruder. We present an example of the application of this work in Section 6.5, where we illustrate how internalising agents allows us to capture parallel attacks and how these attacks are represented within our CSP protocol model.

6.1 Just-in-time principle

In this section, we introduce a protocol model property, referred to as the *just-in-time* property. This property allows us to derive and justify finite bounds upon the intruder that prevent him from requesting an unbounded supply of fresh values, without weakening our analysis (namely, not losing any attacks). We firstly introduce a simple definition of equivalence that we will be using throughout this chapter.

Definition 6.1 (External equivalence) *We say that two traces ω and ω' are "externally equivalent" precisely when*[1]*:*

$$w \upharpoonright \{\mid send,\ receive \mid\} \quad = \quad w' \upharpoonright \{\mid send,\ receive \mid\}$$

∎

Thus, if two traces ω and ω' are externally equivalent, then all behaviour involving external agents is identical in both traces; the only difference between them that may arise, is regarding the internal behaviour of the intruder (either on behalf of himself or the internal agents).

Definition 6.2 (Just-in-time) *Suppose we have a CSP protocol model with a number of externally modelled agents, together with an internal agent S, where S introduces fresh values of some type T.*

We say that a fresh value t of type T, received by an external agent, is generated "just-in-time" precisely when t is freshly introduced (via the corresponding generation of S) after all the protocol messages that precede the receipt of t (in some message M) by the external agent.

We say that S satisfies the "just-in-time" property with respect to type T precisely when, for every trace ω in the system, either (i) all values of type T are generated just-in-time in ω, or (ii) there exists another trace ω' in the system such that ω' is externally equivalent to ω and all values of type T received by an external agent are generated just-in-time in ω'.

∎

[1] Suppose ω is a trace and X is a set of events. $\omega \upharpoonright X$ returns the trace of events in ω that are also members of X.

Notice that this property is concerned only with those fresh values that *are* eventually passed on to external agent processes and the point at which *they* are generated; the fact that the intruder can store fresh values that he never passes on to external agent processes is an issue we discuss later on in this chapter. Intuitively, this means that if the just-in-time property holds in a given CSP protocol model, then there is no advantage to be gained by the intruder to store this type fresh values, unknown to any external agent processes, that will only be introduced into the network later on in the trace.

On the other hand, if a CSP protocol model does not satisfy the just-in-time property, then there exists some trace ω that relies on the intruder being able to store fresh values before the point at which they are passed on to an external agent process. By doing this, the intruder is able to construct and send out messages using these values in a way that cannot be reproduced just-in-time. Clearly, this type of behaviour cannot be discarded or ignored, since it might be crucial towards an attack upon the protocol being modelled.

We consider two simple examples to illustrate our just-in-time property.

Example 6.1 (Satisfying the just-in-time property) *Recall the protocol description introduced earlier in this chapter:*

$$\begin{array}{lll} \textit{Message 1.} & A \rightarrow S & : \{B,\ n_a\}_{SKey(A)} \\ \textit{Message 2.} & S \rightarrow A & : \{n_a,\ k_{ab}\}_{SKey(A)} \end{array}$$

where A is an agent introducing the fresh nonce n_a and S is the server introducing the fresh key k_{ab}. If we model S as internal, then its functionality is captured by the generation:

$$\langle k_{ab} \rangle,\ \{B,\ n_a\}_{SKey(A)} \ \vdash\ \{n_a,\ k_{ab}\}_{SKey(A)}$$

Suppose there are 2 instances of A declared as external agent processes, all given the identity Alice. Consider the following valid sequence of events:

- *Message 1.* $Alice_1 \ \rightarrow\ I_{Server}\ :\ \{Bob, N_1\}_{SKey(Alice)}$
- *Message 1.* $Alice_2 \ \rightarrow\ I_{Server}\ :\ \{Bob, N_2\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_1 \rangle,\ \{Bob, N_1\}_{SKey(Alice)} \vdash \{N_1, K_1\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle,\ \{Bob, N_2\}_{SKey(Alice)} \vdash \{N_2, K_2\}_{SKey(Alice)}$

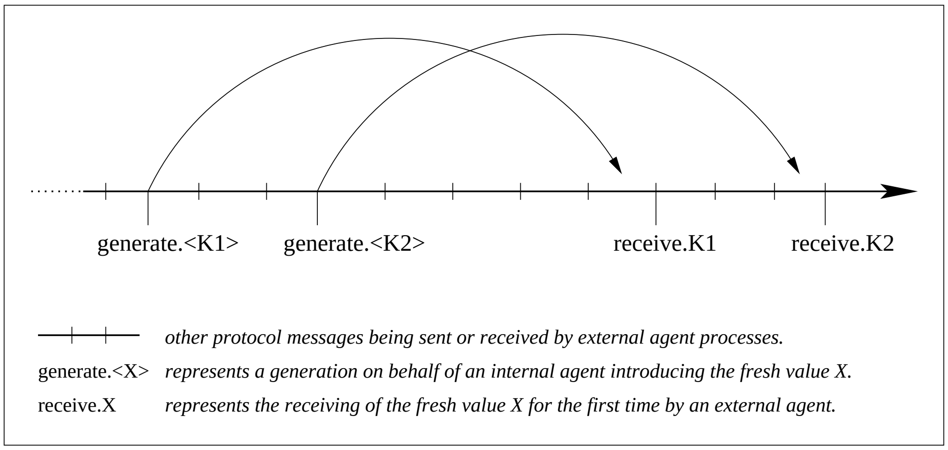

Figure 6.1: Applying the just-in-time property.

- *Message* 2. $I_{Server} \rightarrow Alice_1 : \{N_1, K_1\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice_2 : \{N_2, K_2\}_{SKey(Alice)}$

where I_{Server} represents the intruder acting on behalf of S.

In this trace, the intruder is not generating the message 2's just-in-time; in particular, K_2 is not generated just-in-time. However, we can construct an equally valid trace of the system that does and is externally equivalent to the trace above (illustrated in Figure 6.1):

- *Message* 1. $Alice_1 \rightarrow I_{Server} : \{Bob, N_1\}_{SKey(Alice)}$
- *Message* 1. $Alice_2 \rightarrow I_{Server} : \{Bob, N_2\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_1 \rangle,\ \{Bob, N_1\}_{SKey(Alice)} \vdash \{N_1, K_1\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice_1 : \{N_1, K_1\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle,\ \{Bob, N_2\}_{SKey(Alice)} \vdash \{N_2, K_2\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice_2 : \{N_2, K_2\}_{SKey(Alice)}$

■

Example 6.1 illustrates how the intruder has the ability legitimately to generate many messages on behalf of the server S without necessarily passing them on immediately; the number of fresh values he can request is dependent on the number available. In this particular case, there is no advantage to be gained by the intruder from performing generations early and storing the fresh values; it does not enable him to perform any deductions or further generations towards constructing new messages that he otherwise would not be capable of. The protocol only introduces a single fresh key (encrypted under a public key) per run by the internal agent. At any point, he can only ever generate messages of that form and pass at most one fresh value onto an external agent per run. Furthermore, any deductions that he was able to perform earlier, he is always able to perform in the future (since deductions are never disabled).

However, as soon as we start looking at larger protocol examples, determining whether a given protocol satisfies the just-in-time principle is much less intuitive and often very complex. Example 6.2 gives an example of a protocol that does not satisfy our property.

Example 6.2 (Violating the just-in-time property) *Consider a protocol defined by the following sequence of messages:*

$$\begin{array}{lll}
\textit{Message 1.} & A \to B & : \{k_1,\ A\}_{PK(B)} \\
\textit{Message 2.} & A \to S & : \{B,\ i_a\}_{SKey(A)} \\
\textit{Message 3.} & S \to B & : \{k_2,\ k_3,\ i_a\}_{SK(S)} \\
\textit{Message 4.} & B \to S & : \{n_{sec},\ i_a\}_{k_2} \\
\textit{Message 5.} & A \to B & : \{n_{pub},\ i_a\}_{k_1} \\
\textit{Message 6.} & B \to A & : n_{pub}
\end{array}$$

where k_1, k_2 and k_3 are keys introduced freshly by the agent A and the server S respectively; i_a is an index value introduced freshly by A; and finally, n_{sec} and n_{pub} are fresh nonces introduced by B and A respectively. Note that the encryption in message 3 is with S's secret key, denoting S's signature.

Suppose S is modelled as internal; its functionality is captured by the generation:

$$\langle k_2,\ k_3\rangle,\ \{B,\ i_a\}_{SKey(A)} \ \vdash\ \{k_2,\ k_3,\ i_a\}_{SK(S)}$$

Furthermore, suppose that the system is composed of one external instance each of A and B with identities $Alice$ and Bob respectively.

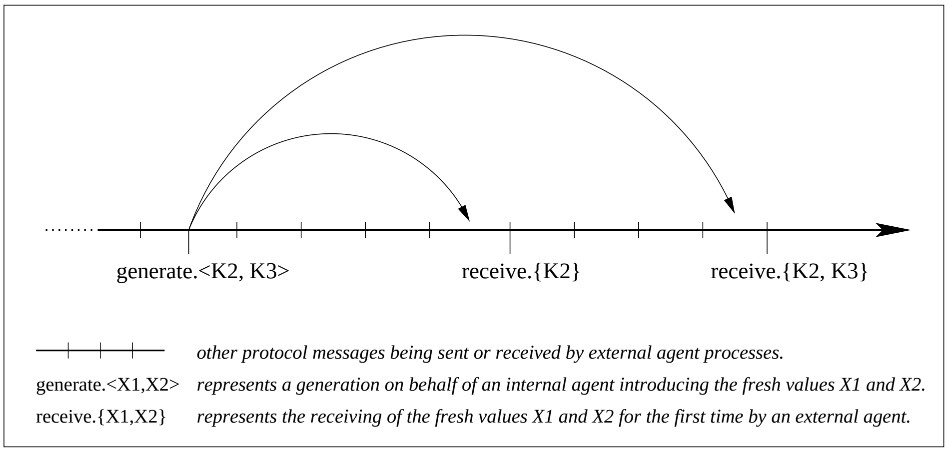

Figure 6.2: Violating the just-in-time property.

Consider the following valid sequence of events:

(1) *Message* 1. $Alice \rightarrow I_{Bob} : \{K_1, Alice\}_{PK(Bob)}$

(2) *Message* 2. $Alice \rightarrow I_{Server} : \{Bob, I_1\}_{SKey(Alice)}$

(3) *(Generation)* $\langle K_2, K_3\rangle,\ \{Bob, I_1\}_{SKey(Alice)} \vdash \{K_2, K_3, I_1\}_{SK(Server)}$

(4) *Intruder deduces* $\{K_2, Alice\}_{PK(Bob)}$ *using message generated in* (3).

(5) *Message* 1. $I_{Alice} \rightarrow Bob : \{K_2, Alice\}_{PK(Bob)}$

(6) *Message* 3. $I_{Server} \rightarrow Bob : \{K_2, K_3, I_1\}_{SK(Server)}$

The message 3 generated by the intruder on behalf of the server (in step (3)*) does not satisfy the just-in-time property in this trace, since he does not immediately pass* K_3 *on to an external agent. Instead, he deliberately chooses to gather it first and use the fresh key* K_2 *to construct a new message* 1*. The intruder then sends this message to Bob in step* (5)*, pretending to be from Alice. It is only at this point that he sends the server-message generated earlier, to Bob in step* (6)*.*

In order to construct an externally equivalent trace that satisfies the just-in-time property, we need to be able to construct a trace such that the generation of the server-message is delayed until immediately before it is *sent on to Bob; in other words, after step* (3) *and before step* (6). *However, this is not possible since the intruder specifically wants the fresh value bound to* k_1 *in message 1 to be the same as that bound to* k_2*; in this trace he uses the* K_2 *generated on behalf of the server, as illustrated in Figure 6.2. We see below how this leads to an attack.*

■

Example 6.2 illustrates how there are cases where it is advantageous for the intruder to store fresh values, unknown to any external agents, that he will only pass on to the network later in the trace (thereby violating the just-in-time principle). The mere fact that storing fresh values gives the intruder an advantage means that it is impossible to find an *externally equivalent* trace where all the messages are generated just-in-time.

Such cases arise when the intruder can exploit dependencies between fresh values and the ways in which they are used within the protocol description. In our example above, the intruder exploits the fact that he can bind the same fresh value to the variables k_1 in message 1 and k_2 in message 3, by generating a fresh server-message 3 early on and using the fresh value K_2 in a new message 1. The fact that the values bound to these two variables are the same, allows him to attack the protocol by the sequence of steps taken above, together with the following subsequent behaviour:

(7)	Message 4.	Bob	$\rightarrow$	I_{Server}	:	$\{N_1,\ I_1\}_{K_2}$
(8)	Message 5.	I_{Alice}	$\rightarrow$	Bob	:	$\{N_1,\ I_1\}_{K_2}$
(9)	Message 6.	Bob	$\rightarrow$	I_{Alice}	:	N_1

By sending the final message 6, *Bob* believes that N_1 is a secret nonce shared only between himself and the server. As we see in step (9), the intruder learns this secret.

In order to perform this attack, the intruder needs to be able to replay message 4 received from *Bob* in step (7) as a message 5 in step (8). The intruder does this by ensuring that the fresh keys bound to k_1 and k_2 are the same. However, the only way he can achieve this, is by generating the server-message 3 *before* sending the corresponding message 1 to *Bob*. In turn, this means that while K_2 is generated just-in-time (since it is immediately passed on to *Bob* as part of the message 1), the fresh value K_3 bound to k_3 is not.

There is no way that the intruder can generate K_3 just-in-time and bind the same fresh value to the variables k_1 and k_2.

Being able to determine whether a protocol model satisfies our just-in-time property is not straightforward; we discuss how we achieve this later on in this chapter. However, once we have established that this property is satisfied by a given protocol model, we are able to derive bounds upon the intruder that prevent him from requesting an unbounded number of fresh values through internal agents' generations and justify that no behaviour of the system is lost. The following section discusses how our just-in-time principle is used for this purpose.

6.2 Constructing a reduced protocol model

In this section, we present an extension to our CSP protocol models that involves introducing special values, referred to as *dummy* values, into the data independent types being generated. Together with the just-in-time property we introduced in the previous section, this extension often enables us to map a protocol model with an infinite supply of fresh values to an *equivalent* reduced system with a finite source of fresh values. By equivalent, we mean that no attacks are lost through the mapping; thus, if no attack is found upon the reduced system, then we can conclude that none exists upon the original infinite version of the system.

6.2.1 Dummy values

The just-in-time property is concerned only with the fresh values that are generated by the intruder and passed on to external agents; it says nothing about any other fresh values he generates and never sends out. We will refer to this latter class of values as *internal fresh values*. It may initially seem rather odd that the intruder would want to generate fresh values and then never pass them on to any external agent processes. However, without placing any restrictions upon the number of fresh values he can request and having an infinite supply of them, the intruder is free to do and behave how he pleases. There are two main reasons why the intruder may want to store internal fresh values. The first is simply because he is able to do so and therefore stores them with no particular gain (in terms of enabling deductions). The intruder can do this, for example, by using the same antecedent to generate many different resulting messages, each characterised by distinct fresh values.

The second motivation for storing internal fresh values is that it might enable further generations and deductions for the intruder to take advantage of and construct new messages that he could not have built otherwise. This ability to internally manipulate messages and construct every possible valid message is crucial when working towards developing a complete analysis of protocols, since it considers the full range of the intruder's abilities towards attacking them.

Distinguishing between these two *classes* of fresh values in any given trace is the key to how we extend our CSP models and justify finite bounds upon the intruder. The observation we make is that it does not actually matter *which* internal fresh values are supplied; what is important is that they exist in some form for the purposes of allowing the intruder to perform whatever manipulations he needs in order to construct the necessary messages. It is not even necessary for these values to be *fresh*, since they are never passed on to external agents. The intruder could just as easily perform subsequent deductions and generations with any values that were strictly for internal use only.

Based on this observation, we introduce a new class of values, referred to as *dummy* values, that will be added to the data independent types being generated. These extra values have the special characteristic that they are not accepted as genuine by any honest process (so the latter will never accept any message involving one). The intruder can use these values itself like any others, in particular doing deductions involving them. The trick is that we allow the intruder to perform, at any time, a "generation" based on a valid input set X, but unless the number of fresh values he is currently storing is less than the given bound, the result will always be based on a dummy value; otherwise, the result may be either a fresh or dummy value. Hence, for a given bound N upon the intruder, we allow the intruder to perform a generation (t, X, Y) for a given input set X and a *fresh* value t precisely when the intruder stores fewer than N fresh values unknown to any external agent processes. In Section 6.4, we discuss what these bounds should be for various classes of protocols.

As an example, consider the *generation* presented in (6.1), representing the functionality of the internal server S. Let's assume that the bound we have derived upon the intruder, in terms of the number of fresh keys he is allowed to store, unknown to any external agents, is 1. If no such generations have been performed yet, then the intruder is free to perform one and request for a fresh key to be bound to k_{ab}. For example, if the intruder intercepted the message 1 $\{Bob,\ N_A\}_{SKey(Alice)}$, then he could perform the following generation:

$$\langle K_1 \rangle,\ \{Bob,\ N_A\}_{SKey(Alice)} \ \vdash\ \{N_A,\ K_1\}_{SKey(Alice)}$$

where K_1 is a fresh key supplied by the Key Manager. However, if the intruder wishes to perform another generation *before* he passes the resulting message 2 onto an external agent process, then the value bound to k_{ab} can only be the dummy value. For example, using the same message 1, the intruder would only be allowed to perform the following generation:

$$\langle K_D \rangle,\ \{Bob,\ N_A\}_{SKey(Alice)} \ \vdash\ \{N_A,\ K_D\}_{SKey(Alice)}$$

where K_D is the dummy value. As we mentioned above, the honest agent processes will not accept the dummy value; its sole purpose is for the benefit of the intruder within his deductive system.

Thus, if the intruder needs N internal fresh values acquired through the various generations he performs, for the sole purpose of constructing some message M introducing another fresh value v, then instead of supplying him with N fresh values, we supply him with the dummy values (other than v, which needs to be fresh). This allows him to accomplish the construction of M without wasting N fresh values.

In practice, we typically declare one dummy value per generated data independent type. However, there is the possibility that this technique introduces false attacks. An example would be where the value being introduced is a key K, and one of the messages contains something encrypted under K that the intruder would not otherwise learn; representing K by the dummy value, which the intruder could learn from elsewhere, would allow him to deduce the contents of the message, as a false attack. A solution, as with the *background* values, is to use two dummy values: one that is created in circumstances where we would expect the intruder to learn it legitimately, and one that is created in other cases. However, a single value appears to suffice more frequently than in the analogous case of background values.

6.2.2 Constructing an equivalent reduced model

Given a protocol model that satisfies the just-in-time property and has an infinite supply of fresh values of the data independent types generated, we can construct an equivalent reduced model where there is only a finite source of fresh values and that incorporates the dummy values for the relevant types. By equivalent, we mean that for every trace in the original infinite model, we can find an externally equivalent trace in our reduced model. Such a trace is constructed by mapping all the internal fresh values to the dummy values, leaving only those fresh values in the trace that are passed on to external agent processes (and therefore generated just-in-time). Proposition 6.1 captures this more formally.

Proposition 6.1 *Suppose System*(AS) *is a protocol model with the set of roles* AS*, where some role* A *in* AS *is modelled as internal and introduces fresh values of some data independent type* T*. Suppose further that System*(AS) *is provided with an unbounded supply of fresh values of type* T*. If System*(AS) *satisfies the just-in-time property, then we can construct a reduced system System*$_R(AS)$ *such that, for every trace* ω *in System*(AS)*, there exists a trace* ω' *in System*$_R(AS)$ *where* ω *and* ω' *are externally equivalent. System*$_R(AS)$ *is constructed as follows:*

1. *The dummy value strategy is implemented (as described above) for type* T*.*
2. *The maximum number of fresh values of type* T *the intruder can store (unknown to any external agents) is equal to the maximum number of them he can pass on to an external agent in a protocol message.*

Proof We need to show that every trace ω in *System*(AS) can be mapped to an externally equivalent trace ω' in *System*$_R(AS)$, where *System*$_R(AS)$ is subjected to the conditions presented above. This is quite straightforward by the definition of just-in-time, as follows.

By definition, the fact that *System*(AS) satisfies the just-in-time property (with regards to the internal agent A and type T) means that, for every trace ω in the system, either (i) all values of type T are generated just-in-time, or (ii) there exists another trace ω' in the system such that ω' is externally equivalent to ω and all values of type T are generated just-in-time in ω'. We can, therefore, consider solely these just-in-time traces and still capture all possible behaviour (by the definition of external equivalence). This is the first step of the reduction. The supply and storage of fresh values though are still infinite. We now need to justify reducing this system further to *System*$_R(AS)$, by showing that every trace in this subset of traces of $System(AS)$ can be mapped to an externally equivalent trace in $System_R(AS)$.

Suppose ω is a trace of the reduced (but still infinite) $System(AS)$, where all the values of type T are generated just-in-time (by definition of the reduction step above). We construct an externally equivalent trace ω' in $System_R(AS)$ as follows. All values of type T, except for the ones generated just-in-time and therefore being passed on to external agents immediately, are mapped to the dummy value D_T. Since the remaining fresh values are generated just-in-time, the maximum number of them the intruder will ever need to store (unknown to any external agents) is the maximum number of them he can pass on to an external agent for the first time in some protocol message. Therefore the bound we place upon the intruder in Condition 2 of $System_R(AS)$ is justified. Furthermore, the fact that there are only ever a finite number of external agents declared in our CSP models, means that $System_R(AS)$ will only need a finite source of fresh values of type T (by the mechanics of the recycling mechanism in our models, together with the assumption that agent processes never remember fresh values for more than a single run).

■

We illustrate this mapping more intuitively in the following example.

Example 6.3 *Consider a simple protocol defined as follows:*

$$\begin{array}{llcl} \text{Message 1.} & A \rightarrow S & : & \{n_a\}_{SKey(A)} \\ \text{Message 2.} & S \rightarrow A & : & \{k_1,\ n_a\}_{SKey(A)} \\ \text{Message 3.} & S \rightarrow A & : & \{k_2,\ n_a\}_{SKey(A)} \end{array}$$

where n_a is a fresh nonce introduced by A, k_1 and k_2 are keys supplied freshly by the server S and $SKey(A)$ is a symmetric key known only by S and A.

Consider the following trace of events that can take place, where the server is modelled as internal, there is one instance of A declared externally with identity Alice and no dummy values are implemented:

- *Message 1.* $Alice \rightarrow I_{Server} : \{N_A\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \vdash \{K_1, N_A\}_{SKey(Alice)}$

- *(Generation 2)* $\langle K_2 \rangle, \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_1, N_A\}_{SKey(Alice)} \end{array} \right\} \vdash \{K_2, N_A\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice : \{K_2,\ N_A\}_{SKey(Alice)}$
- *Message* 3. $I_{Server} \rightarrow Alice : \{K_1,\ N_A\}_{SKey(Alice)}$

By storing the two server-generated messages 2 and 3, the intruder is able to replay them in reverse order. This trace does not conform to the just-in-time property, since K_1 is not generated just-in-time. However, there exists an externally equivalent trace in this model that does, for example:

- *Message* 1. $Alice \rightarrow I_{Server} : \{N_A\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_3 \rangle,\ \{N_A\}_{SKey(Alice)} \vdash \{K_3, N_A\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle, \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_3, N_A\}_{SKey(Alice)} \end{array} \right\} \vdash \{K_2, N_A\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice : \{K_2,\ N_A\}_{SKey(Alice)}$
- *(Generation 3)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \vdash \{K_1, N_A\}_{SKey(Alice)}$
- *Message* 3. $I_{Server} \rightarrow Alice : \{K_1,\ N_A\}_{SKey(Alice)}$

where the fresh key K_3 is not subsequently used, but exists for the sole purpose of allowing the intruder to gain access to a message 3 from the server to replay as a message 2 to $Alice$. The generation for a message 2 can be performed again with the same antecedents and another fresh key (K_1) just-in-time for supplying it as a message 3 to $Alice$.

Finally, if we implement the notion of dummy values, we can map the previous trace to the following corresponding one in the reduced model:

- *Message* 1. $Alice \rightarrow I_{Server} : \{N_A\}_{SKey(Alice)}$
- *(Generation 1)* $\langle K_D \rangle,\ \{N_A\}_{SKey(Alice)} \vdash \{K_D, N_A\}_{SKey(Alice)}$
- *(Generation 2)* $\langle K_2 \rangle, \left\{ \begin{array}{l} \{N_A\}_{SKey(Alice)}, \\ \{K_D, N_A\}_{SKey(Alice)} \end{array} \right\} \vdash \{K_2, N_A\}_{SKey(Alice)}$
- *Message* 2. $I_{Server} \rightarrow Alice : \{K_2,\ N_A\}_{SKey(Alice)}$

- *(Generation 3)* $\langle K_1 \rangle,\ \{N_A\}_{SKey(Alice)} \vdash \{K_1, N_A\}_{SKey(Alice)}$
- *Message* 3. $I_{Server} \rightarrow Alice : \{K_1,\ N_A\}_{SKey(Alice)}$

■

Example 6.3 provides a useful and simple example of how we can map a trace in the original protocol model (with potentially infinite supply of fresh values) to an externally equivalent one in the reduced model. In this particular example, the intruder needed to perform a generation with a dummy value in order to gain access to the second server-message, before passing on the first. Since intruder deductions and generations are never disabled, he can simply use the same antecedents (in this case, $\{N_A\}_{SKey(Alice)}$) to generate another server-message 3 just-in-time. Which actual fresh value gets supplied does not matter (since the type is data independent), as long as it is fresh. Thus, this has the same effect as gathering the two messages and playing them in reverse order.

To summarise thus far, we have shown how we can map a protocol model that satisfies the just-in-time property and has an infinite supply of fresh values to an equivalent reduced model with a finite source of fresh values. This was achieved by extending our CSP models with special values known as dummy values. These values are used instead of the potentially infinite number of internal fresh values the intruder uses to perform all of his desired internal activities and manipulations.

As discussed in the introduction of this chapter, a protocol model with an infinite supply of fresh values enables the intruder to perform attacks for any degree of parallelism among the internal agents. By being able to map an infinite model with an internal agent A to an equivalent reduced one (for protocols that satisfy the just-in-time property), means that we are able to capture attacks upon protocols for any degree of parallelism within A by performing a finite refinement check.

There are two main questions that we still need to consider. The first is how we determine whether a given protocol model satisfies the just-in-time property. The second is, given a protocol model that satisfies this property and makes use of the dummy values, what should the finite bound upon the intruder be to prevent him from using fresh values for internal purposes only, while still ensuring that he can perform all possible interactions with the external agent processes? We address these questions in the remaining part of this chapter.

6.3 Factorisability of internal agents

We now introduce a new property, referred to as the *factorisability* of internal agents, and show that if a given internal agent S within a protocol model can be *factorised* in this way, then we can apply the just-in-time principle to justify the bounds placed upon the intruder with regards to the fresh values introduced on behalf of S (via the corresponding generations).

The factorisability property is quite a restrictive one and there are many protocol examples that do not satisfy it. However, it has proved to be extremely useful towards determining and justifying finite bounds upon the intruder with regards to the number of fresh values he can request through internal agent generations. Firstly, we show that if the internal agents within a protocol model satisfy this property, then our just-in-time property is also satisfied. Secondly, it is simpler to derive a class of protocols that are factorisable than determining whether the just-in-time argument holds directly.

Definition 6.3 (Factorisability) *We say that an internal agent A is factorisable with respect to some data independent type T precisely when, for each run R of A that generates fresh values $v_1, \ldots, v_k$ of type T, the following conditions are satisfied:*

1. *We can construct runs $R_1, \ldots, R_k$ of A, where each run R_i contains the fresh value v_i and the dummy value only.*
2. *For each output message M in R, there exists at least one R_i that contains M, where $i \in \{1, \ldots, k\}$.*
3. *For all v_1 and v_2 of type T generated on behalf of A, if A receives v_1 back in some protocol message, then no subsequent message sent or received by A contains v_2.*

$\blacksquare$

An intruder involving an internalised factorisable agent A is equivalent to one in which A has been replaced by A', where A' can only deliver one fresh value per independent run.

Thus, the act of the intruder storing up internal agent messages (and therefore fresh values) in the model that has an unbounded number of fresh values

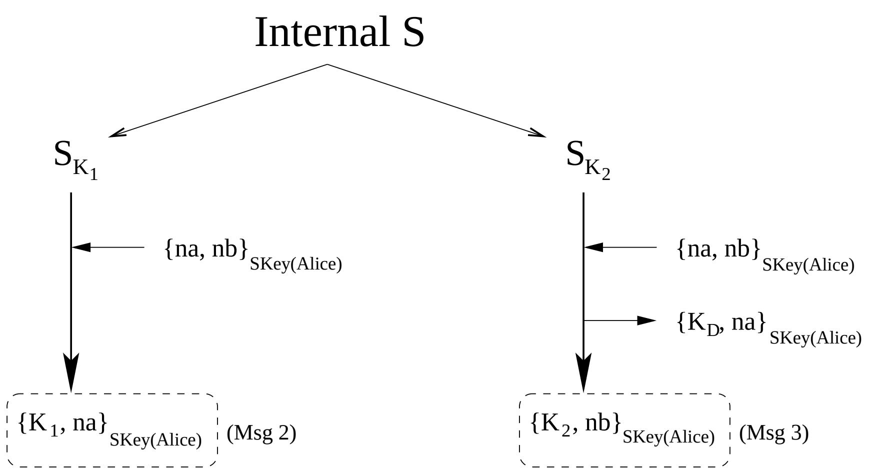

Figure 6.3: *Factorisable* internal server S in Example 6.4.

(and no dummy values) to allow him to replay them in a different order, is simulated here by the intruder being able to perform independent runs with the server for each message (and fresh value) required. By the definition of factorisability, he can achieve this by replaying external agent messages as input to the various internal agent runs he is interested in, since there exists a run for each internally generated message and fresh value, where all the other fresh values generated are dummy values.

The fact that the other values generated are dummy values means that the generation of any message containing a fresh value on behalf of an internal agent A is not dependent on A being able to distinguish the runs of the protocol; otherwise one could not replay the same messages as input to A. Hence the need for the 3^{rd} condition in the definition; we discuss this condition later Example 6.6.

To illustrate the factorisability property more clearly, consider the following example of a simple protocol whose internal server is factorisable.

Example 6.4 (A factorisable internal server) *Consider the following protocol:*

$$
\begin{array}{lll}
\textit{Message 1.} & A \rightarrow S & : \{n_a,\ n_b\}_{SKey(A)} \\
\textit{Message 2.} & S \rightarrow A & : \{k_1,\ n_a\}_{SKey(A)} \\
\textit{Message 3.} & S \rightarrow A & : \{k_2,\ n_b\}_{SKey(A)}
\end{array}
$$

where n_a and n_b are two nonces introduced freshly by agent A and k_1 and k_2 are two keys introduced fresh by the server S.

S modelled as internal is factorisable, since it satisfies the conditions required in Definition 6.3. As illustrated in Figure 6.3, it is straightforward to see that we can factor the runs of S such that each run only ever produces a single fresh value (the rest being dummy values). Since there are only the two messages generated on behalf of S where a single fresh value is introduced in each, every possible output message of S can clearly be generated in one of the independent runs of S. Furthermore, S does not depend on receiving any fresh values previously introduced (by S itself), thereby satisfying the 3^{rd} requirement of the factorisability definition.

■

Example 6.4 provides a simple example of a factorisable internal server S and as a consequence, how the intruder can gain access to the fresh values bound to k_1 and k_2 by performing independent runs with S. He achieves this using the dummy values (as illustrated in Figure 6.3) and the fact that, by definition, he is able to simply replay the external agent A's message 1 to S for each run. Without the use of dummy values, the intruder could achieve the same result by performing the same independent runs, where the dummy values are replaced by freshly supplied ones.

There are many protocols however, whose internal agents are not factorisable. To provide further intuition of this, we present two examples of such protocols, each violating the factorisability property for different reasons.

Example 6.5 (Non-factorisable internal agents) *Consider a protocol with the following type of message sequence:*

$$
\begin{array}{lll}
\textit{Message } 1. & S \to \ldots & : \{k_1\}_E \\
\textit{Message } 2. & S \to \ldots & : \{k_2\}_E \\
\quad\vdots & & \\
\textit{Message } n. & S \to \ldots & : \{k_1,\ k_2\}_E
\end{array}
$$

where S is the internal server introducing the fresh keys k_1 and k_2. E is some encryption key known to S and whomever the messages are intended for; the type of encryption is not relevant for the purposes of this example.

If S is factorisable, then we need to be able to construct independent runs for each fresh value introduced by S. Furthermore, every output message from

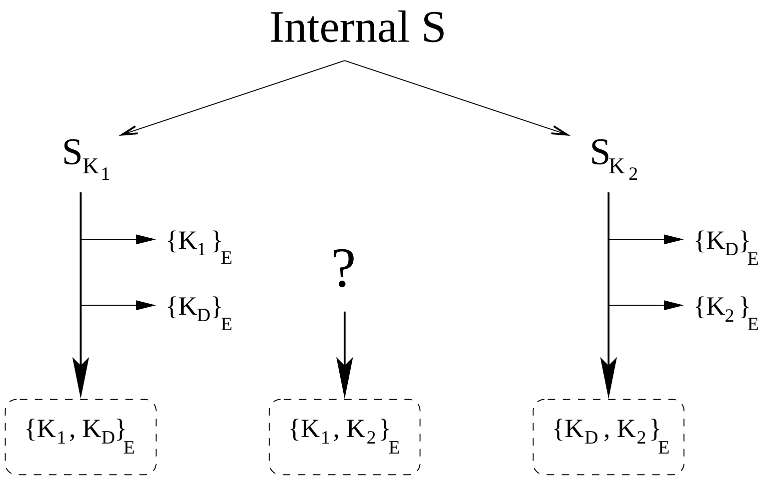

Figure 6.4: *Non-factorisable* internal server S in Example 6.5.

S must be present in one of these runs. In this example, we must be able to factor S in two runs, namely one where k_1 was bound to a fresh value and k_2 was bound to the dummy value, and the other vice versa. If we gain the fresh values K_1 and K_2 bound to k_1 and k_2 respectively in separate runs (while the other is bound to the dummy value), S will either generate a message n of the form $\{K_1, dummy\}_E$ or $\{dummy, K_2\}_E$; the message $\{K_1, K_2\}_E$ is not generated in any of the factored runs. Figure 6.4 illustrates these two server runs and the resulting problem.

■

Example 6.5 highlights the problem that can arise as a result of having multiple fresh values in a single message generated by an internal agent, even though these values were both introduced in previous messages. In order to satisfy the factorisability property, there cannot be any *dependencies* among fresh values introduced by an internal agent S of this nature, since they could result in S being able to distinguish between his runs, thereby making it impossible to factor S according to our definition.

Dependencies between fresh values such as these are not the only source of problems when it comes to factorising internal agents. A more subtle way an internal agent S can distinguish between runs and potentially prevent S from being factorised, is if we allow S to generate a fresh value that depends on S receiving a previously introduced fresh value. The effect this can have is illustrated in Example 6.6.

Example 6.6 (Not factorisable) *Consider the following protocol:*

$$
\begin{array}{llcl}
\textit{Message 1.} & S \rightarrow A & : & \{k_1\}_{SKey(A)} \\
\textit{Message 2.} & A \rightarrow S & : & \{k_1,\ n_a\}_{SKey(A)} \\
\textit{Message 3.} & S \rightarrow A & : & \{k_2,\ n_a\}_{SKey(A)} \\
\textit{Message 4.} & S \rightarrow A & : & \{k_3,\ n_a\}_{SKey(A)}
\end{array}
$$

where k_1, k_2 and k_3 are fresh keys introduced by S, n_a is a fresh nonce introduced by A, SKey(A) is a previously established key known only to A and S. S modelled as an internal server is not factorisable, since the messages 3 and 4 rely on S receiving the previously introduced fresh value k_1. This violates the 3^{rd} condition of the factorisability definition.

■

Example 6.6 illustrates why we need the 3^{rd} condition of the factorisability definition. For the internal server S to be factorisable, we must be able to construct an independent run R_i of S for each fresh value v_i generated by S, where all other fresh values (generated by S) are dummy ones. This is not possible for the following reason. Consider the case where the intruder has intercepted the following message 2: $\{K_1, N_A\}_{SKey(Alice)}$ from the external agent *Alice* and wants to generate corresponding messages 3 and 4 on behalf of S in independent runs. Thus, there will be one run where a fresh value is bound to k_3 (in message 4) and the dummy value bound to k_2 (in message 3) and another run vice versa. To ensure that both messages generated by S contain the corresponding fresh nonce N_A, the intruder must prompt S with the same message 2 in both runs. This is precisely where the problem lies.

The server S will only ever accept a message 2 that corresponds to a message 1 sent by S and thus contains the correct fresh key bound to k_1. Furthermore, S will never introduce the same fresh key in a message 1, since our model ensures that all fresh values are indeed fresh and distinct from each other. Therefore, replaying the same message 2 to S is not an option for the intruder. If the intruder performs a second run with S and engages another external instance of *Alice* (as role A) by sending a fresh message 1, then the resulting message 3 or 4 generated will not contain the original nonce N_A.

Hence, S cannot be factorised because he generates messages containing fresh values that are dependent on S receiving previously introduced fresh values. Once again, the problem arises as a result of subtle dependencies between the generation of the fresh values.

The motivation for introducing the factorisability property was to help determine which protocols satisfy the just-in-time property. The relationship between these two properties is captured by Proposition 6.2.

Proposition 6.2 *Suppose we have a CSP protocol model with a number of externally modelled agents, together with an internal agent S, where S introduces fresh values of type some T. If S is factorisable, then S satisfies the just-in-time property.*

Proof For S to satisfy the just-in-time property, it must be the case that, for every trace ω in the system, we can construct an equally valid trace ω' of the system such that ω' is externally equivalent to ω and all the values of type T are generated just-in-time in ω'.

Suppose ω is a trace of our system (with the factorisable internal agent S). If all the values of type T are generated just-in-time in ω, then clearly our implication is satisfied. However, if this is not the case, then we can construct another trace ω' that is externally equivalent to ω and satisfies the just-in-time property with respect to T, as follows.

When constructing ω', we need to consider every fresh value of type T in ω that is received by an external agent A (in some message M) for the first time, but arose in a generation of S before the last protocol message preceding the receipt of M by A. Suppose t is such a value, M_t is the message that passes t on to any external agent A for the first time, and MS is the sequence of protocol messages, sent and received by external agents, that occur after the generation of t and before the receipt of M_t by A. Furthermore, let G_t be the generation of S that introduces the fresh value t and takes the form $\langle t \rangle,\ X \ \vdash \ Y$, where X is the antecedent (set of input messages) of this generation, known by the intruder (otherwise he would not be able to perform this generation!), and Y is the set of messages generated as a result.

By definition, we know that no messages in MS contain t (since M_t is the first) and therefore none of them rely on the fact that G_t is introducing a fresh value t (as opposed to a dummy one). Their construction may, nevertheless, rely on G_t taking place; for example, the intruder may use other components in the set Y (resulting from G_t) to construct some of them (either directly or through further deductions being enabled), or prompt deductions or generations of S that follow on from G_t. Thus, when constructing ω' from ω, we cannot necessarily move this generation forward on the trace to satisfy the just-in-time property. We can, however, construct this sub-trace of ω' (with respect to t satisfying our property) as follows. Firstly, we replace the generation G_t with the generation G_{D_T}, whose only difference is that the dummy value D_T (for type T) is supplied instead of the fresh value t; G_{D_T} uses the same antecedent as G_t and therefore takes the form $\langle G_{D_T} \rangle,\, X \,\vdash\, Y'$ (Y' differs from Y above only in that all instances of t are replaced by D_T). Secondly, all instances of t in subsequent deductions and further generations within the intruder, that take place *before* the receipt of M_t by A, are replaced by D_T. Thirdly and finally, by the definition of factorisability, we extend the trace with a new independent run R_t of S after the last message in MS and before the receipt of M_t by A, such that the fresh value required in M_t, namely t, is generated on behalf of S; any other values generated in R_t are bound to the dummy value. The intruder can simply replay the same messages he used earlier on in the trace, to prompt S (through the corresponding deductions and generations of S) to perform R_t. The fact that S is factorisable means that the intruder is always able to achieve this, and therefore generate these fresh values just-in-time.

To construct ω', we simply repeat this process for every fresh value t of type T that is not generated just-in-time in ω.

■

6.4 Bounding the intruder's appetite

In the introduction of this chapter, we raised the following question: How can we reasonably limit the intruder's appetite for fresh values when he has the capacity of requesting any number he wishes on behalf of the internal agents via their corresponding generations? As discussed throughout this chapter, the intruder can achieve this, for example, by performing a generation on behalf of an internal agent an unbounded number of times, using the same set of input messages (as the antecedent) and requesting a fresh supply of values each time. Unless restricted, the intruder can continue doing this, until the respective manager processes run out of fresh values to supply the generations with.

In Section 6.2, we showed how a protocol model that satisfies the just-in-time property and has an infinite supply of fresh values, can be mapped to an equivalent reduced model with a finite source of fresh values. By equivalent, we mean that every trace in the infinite model can be mapped to an externally equivalent one in the reduced model. This was achieved by introducing special dummy values into the reduced models, as discussed in 6.2. Two questions arose from this work. The first was concerned with how we determine whether a protocol model satisfies the just-in-time property; we introduced a further property called factorisability (Section 6.3) that helps address this question. The second was: Given a protocol model that satisfies the just-in-time property and makes use of the dummy values, what should the finite bound be upon the intruder to prevent him from using fresh values for internal purposes only, while still ensuring that he can perform all possible interactions with the external agent processes? Clearly, ensuring that the intruder is not restricted in his interactions with other agents, implies that he is able to find attacks, should they exist, for any degree of parallelism among the internal agents.

In this section, we seek to address the latter question and derive suitable bounds upon the intruder for three classes of protocols. We assume that all the models make use of the dummy value strategy.

Proposition 6.3 *Suppose $System(AS)$ represents the CSP model for some protocol P, where AS is the set of roles in P. Suppose further that we model the role A in AS as internal, where A introduces fresh values of some type T. We claim that if the following conditions are satisfied:*

1. *A introduces at most 1 fresh value of type T per run.*
2. *A protocol message can contain at most 1 value of type T.*
3. *The intruder can store at most 1 fresh value of type T, unknown to any of the external agents.*

then no attack found upon $System(AS)$ implies that no attack exists upon P for any degree of parallelism within A.

Proof As discussed in the introduction of this chapter, if $System(AS)$ was given an infinite supply of fresh values of type T, the intruder would be able to capture any degree of parallelism within the internal agent A and therefore perform attacks, irrespective of the number of instances of A required.

For this proposition to hold, it must be the case that every trace of such an infinite version of this system can be mapped to an externally equivalent trace in the reduced model, defined by the conditions presented above. We prove this by firstly proving that A satisfies the just-in-time property and then justifying how this reduced system is externally equivalent to the corresponding infinite version. To prove the just-in-time property, we make use of our factorisability argument and Proposition 6.2.

A generates one fresh value t per protocol run. Therefore, the generation of t does not depend on A receiving any fresh values previously introduced by A; this satisfies the 3^{rd} requirement of the factorisability property. Furthermore, since A only introduces a single fresh value per run, then it clearly follows that A is factorisable, since we can simply perform a complete and independent run of A, for each fresh value of type T that can be generated on behalf of A. By Proposition 6.2, A therefore also satisfies the just-in-time property with respect to type T.

By Proposition 6.1, we can conclude that the version of $System(AS)$ with an infinite supply of fresh values of type T can be reduced to an equivalent one (traces are externally equivalent) with a finite source under the condition placed upon the the intruder, since (i) $System(AS)$ satisfies the just-in-time property, (ii) the dummy value strategy is implemented and (iii) the number of values of type T that the intruder is able to store (unknown to any external agents) is equal to the maximum number of them an external agent can learn for the first time (in this case 1). Therefore, any attack that exists upon the protocol model that supplies A with an infinite number of fresh values of type T (reflecting any degree of parallelism of A) and allows the intruder to store any number of them, can be mapped to an equivalent attack upon the reduced version of the system, where the intruder is bounded by condition 3 above.

■

We weaken Proposition 6.3 to allow internal agents to introduce multiple fresh values in a single run, while still justifying a bound of 1 upon the intruder.

Proposition 6.4 *Suppose System(AS) represents the CSP model for some protocol P, where AS is the set of roles in P. Suppose further that we model the role A in AS as internal, where A introduces fresh values of some type T. We claim that if the following conditions are satisfied:*

1. *A protocol message can contain at most 1 value of type T.*
2. *The progress of A never depends upon the receipt of previously introduced values of type T on behalf of A.*
3. *The intruder can store at most 1 fresh value of type T, unknown to any of the external agents.*

then no attack found upon System(AS) implies that no attack exists upon P for any degree of parallelism within A.

Proof For the same reasons presented in Proposition 6.3, we start by proving that the internal agent A satisfies the just-in-time property and then justify how this reduced system is externally equivalent to the corresponding infinite version. To prove the just-in-time property, we make use of our factorisability argument and Proposition 6.2.

Suppose R is a run of A represented by a sequence of deductions and generations DG_R performed by the intruder on behalf of A, resulting in the set of output messages MS_R. Furthermore, suppose that $t_1, \dots, t_k$ are fresh values generated by the generations $G_1, \dots, G_k$ of A respectively, in R. R can be factored into k independent runs of A, according to the factorisability definition, as follows.

For each fresh value t_i ($\in \{t_1, \dots, t_k\}$) generated by the generation G_i of A, the intruder can construct an independent run R_i on behalf of A, by performing the same sequence of deductions and generations in DG_R, where the fresh value t_i is indeed generated as fresh in G_i and all other generations of A are supplied with the dummy value D_T for type T. Consequently, all instances of t_i in the resulting output messages MS_{R_i} from run R_i will remain the same, whereas all instances of the other fresh values introduced in R on behalf of A will be mapped to D_T. The intruder is able to perform these independent runs on behalf of A by simply re-using the same antecedents each time; this reflects the intruder replaying the same input messages to A, k times. He can choose to perform these runs in any order, either sequentially or interleaved (as illustrated in Example 6.7). We know, from condition *2* above, that A never relies on the receipt of previously introduced fresh values (by A) in order to generate fresh values; therefore, A is not able to distinguish the runs with regards to the input stimuli given.

We must ensure that each output message in MS_R is present in one of the factored runs. This is indeed the case, since by definition (condition *1* above), all protocol messages contain at most one value of type T. Therefore, for each fresh value t_i ($\in \{t_1, \ldots, t_k\}$), the messages in MS_R containing t_i will be generated in run R_i; the fact that the other values are mapped to dummy values, does not affect these output messages (since they only contain one fresh value, namely t_i).

Now that we have established that A is factorisable, we know from Proposition 6.2, that A therefore also satisfies the just-in-time property. The bound of 1 placed upon the intruder, regarding the number of fresh values of type T he can store, unknown to any external agents, is justified for precisely the same argument given in Proposition 6.3 (making use of Proposition 6.1).

■

The following example illustrates more intuitively how, for a given run R of an internal server A, generating a set of output messages MS_R, the intruder can achieve precisely this same set MS_R by performing the corresponding factored runs of A, each one only introducing a single fresh value.

Example 6.7 *Consider a protocol defined as follows:*

$$\begin{array}{lll}
\textit{Message } 1. & A \rightarrow S & : \{n_a\}_{SKey(A)} \\
\textit{Message } 2. & S \rightarrow A & : \{n_a,\ k_1\}_{SKey(A)} \\
\textit{Message } 3. & A \rightarrow S & : \{n_b\}_{SKey(A)} \\
\textit{Message } 4. & S \rightarrow A & : \{n_a,\ n_b,\ k_2\}_{SKey(A)} \\
\textit{Message } 5. & A \rightarrow S & : \{n_c\}_{SKey(A)} \\
\textit{Message } 6. & S \rightarrow A & : \{n_a,\ n_b,\ n_c,\ k_3\}_{SKey(A)}
\end{array}$$

where n_a, n_b and n_c are nonces freshly introduced by agent A, and k_1, k_2 and k_3 are keys freshly introduced by the server S. $SKey(A)$ is a symmetric key shared between S and A only and assumed to be secure.

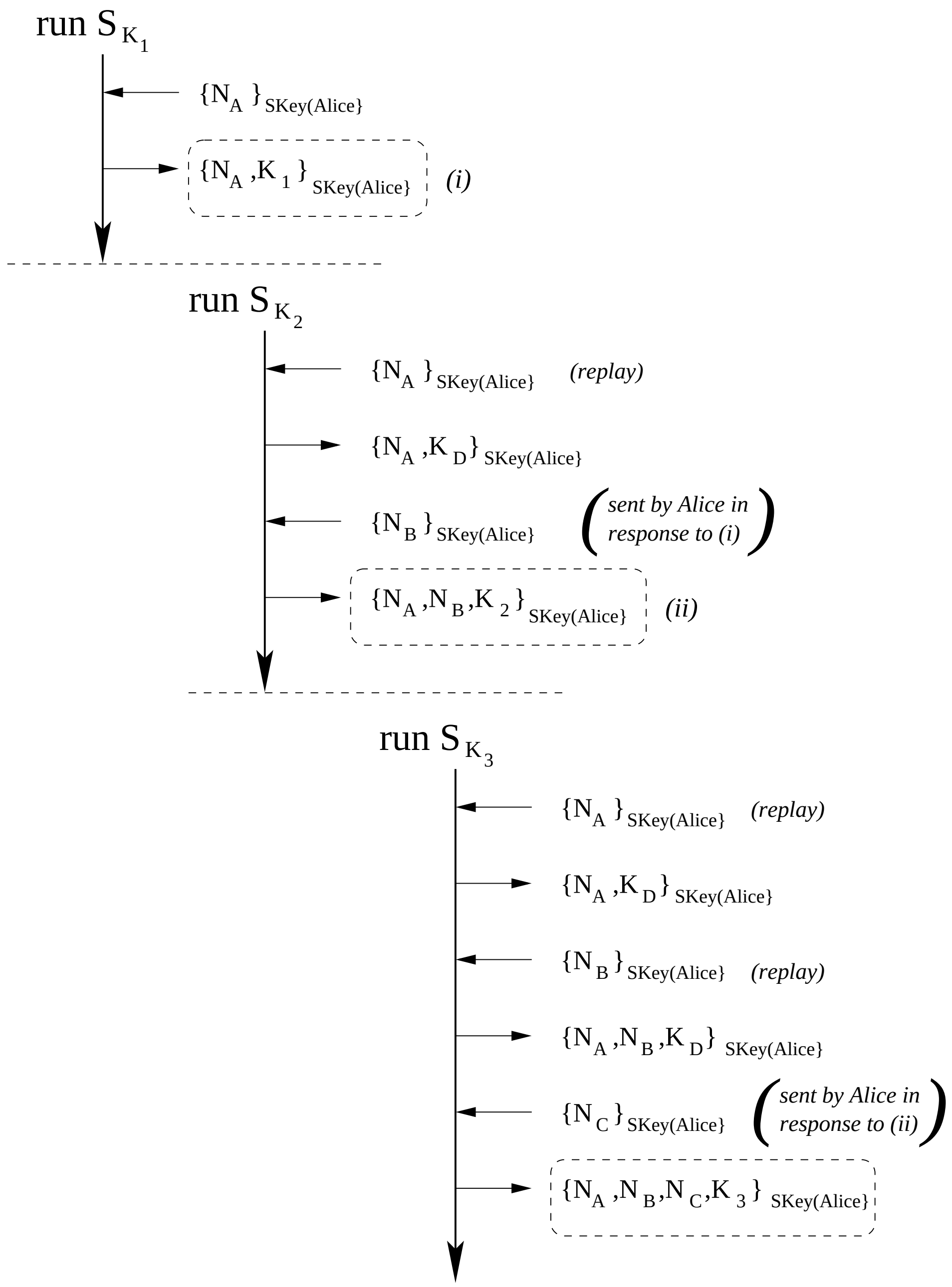

Figure 6.5: Performing the 3 factored runs of S in Example 6.7.

Suppose the intruder performs the following run R on behalf of the internal server S, together with one external instance of A with identity Alice:

$$\begin{array}{lrcl}
\textit{Message 1.} & \textit{Alice} \rightarrow I_{\textit{Server}} & : & \{N_A\}_{\textit{SKey(Alice)}} \\
\textit{Message 2.} & I_{\textit{Server}} \rightarrow \textit{Alice} & : & \{N_A,\ K_1\}_{\textit{SKey(Alice)}} \\
\textit{Message 3.} & \textit{Alice} \rightarrow I_{\textit{Server}} & : & \{N_B\}_{\textit{SKey(Alice)}} \\
\textit{Message 4.} & I_{\textit{Server}} \rightarrow \textit{Alice} & : & \{N_A,\ N_B,\ K_2\}_{\textit{SKey(Alice)}} \\
\textit{Message 5.} & \textit{Alice} \rightarrow I_{\textit{Server}} & : & \{N_C\}_{\textit{SKey(Alice)}} \\
\textit{Message 6.} & I_{\textit{Server}} \rightarrow \textit{Alice} & : & \{N_A,\ N_B,\ N_C,\ K_3\}_{\textit{SKey(Alice)}}
\end{array}$$

where the messages sent by I_{Server} are those generated by the intruder on behalf of the server (we simplify this example by leaving out the actual generation steps). The output messages of the server in R is therefore the set MS_R containing messages 2, 4 and 6.

By Proposition 6.4, we know that the server is factorisable and consequently, the intruder must be able to achieve the same output messages MS_R by running the corresponding factored runs of S, one for each generation of K_1, K_2 and K_3 respectively. This is indeed the case, as illustrated in Figure 6.5.

As we can see in this figure, the intruder can perform the factored run of S (referred to in the figure as S_{K_1}) that generates the fresh key K_1 in message 2. Once he has received the necessary message 3 from Alice, the intruder then replays message 1 to a second factored run of S (S_{K_2}), together with the message 3 received, to generate the corresponding message 4 with the fresh key K_2; the key generated in message 2 in this run is now bound to the dummy key K_D. Finally, the intruder performs a third factored run of S (S_{K_3}) by replaying all the previous input messages to S in R, namely messages 1, 3 and 5 (that was subsequently sent by Alice), to generate the corresponding message 6 with the fresh key K_3; the keys generated in messages 2 and 4 in this run are now bound to the dummy value K_D.

■

This example provides a useful illustration of how the intruder can use factored runs of S, each introducing a single fresh value, to achieve the same result as any standard run of S. Furthermore, it highlights how the intruder may need to perform these factored runs interleaved and at different speeds, depending on the information required in each one. As was the case above, some of these runs may depend on information obtained in others.

In Proposition 6.4, we demand that all protocol messages contain no more than a single value of type T. In terms of satisfying the just-in-time property with regards to the internal agent A, it suffices to ensure that A only ever introduces a single fresh value of type T at any given time. However, we need this stricter condition to justify the resulting bound of 1 upon the intruder. We illustrate this in the following example.

Example 6.8 *Consider a protocol with the following first three messages:*

$$\begin{array}{lll} \textit{Message } 1. & A \rightarrow B & : \ x_1 \\ \textit{Message } 2. & A \rightarrow B & : \ x_2 \\ \textit{Message } 3. & B \rightarrow S & : \ \{x_3,\ x_4\}_{PK(S)} \\ & \vdots & \end{array}$$

where x_1, x_2, x_3 and x_4 are values of some data independent type T introduced freshly by A and B respectively.

Suppose A is modelled as an internal agent. A satisfies the just-in-time property for the same argument presented in the proof for Proposition 6.4. However, a bound of 1 upon the intruder, with regards to type T turns out to be too stringent. Consider the following valid sequence of events:

(1) *The intruder generates the following message 1 on behalf of A:*

$$\langle X_1 \rangle,\ \{\} \ \vdash \ X_1$$

(2) *The intruder generates the following message 2 on behalf of A:*

$$\langle X_2 \rangle,\ \{X_1\} \ \vdash \ X_2$$

(3) *The intruder uses the two fresh values generated on behalf of A to fake the following message to Sam (pretending to be from Bob):*

$$\textit{Message } 3. \quad I_{Bob} \rightarrow Sam \ : \ \{X_1,\ X_2\}_{SKey(Sam)}$$

Although X_1 and X_2 are generated just-in-time, if we place a bound of 1 upon the intruder, then he would not be able to perform this sequence of events (since he stores 2 fresh values unknown to any external agents).

■

We can, however, extend Proposition 6.4 further to allow more general bounds upon the intruder, while maintaining the just-in-time property upon the internal agents.

Proposition 6.5 *Suppose System(AS) represents the CSP model for some protocol P, where AS is the set of roles in P. Suppose further that we model the role A in AS as internal, where A introduces fresh values of some type T. We claim that if the following conditions are satisfied:*

1. *A introduces at most 1 value of type T per message.*
2. *The progress of A never depends upon the receipt of previously introduced values of type T on behalf of A.*
3. *The intruder can store N fresh values of type T, unknown to any of the external agents, where N is the maximum number of values of type T in any single protocol message.*

then no attack found upon System(AS) implies that no attack exists upon P for any degree of parallelism within A.

Proof Internal agent A is factorisable for precisely the same argument given in Proposition 6.4, since by condition 1 above, A introduces at most a single fresh value of type T per message. By Proposition 6.2, A therefore satisfies the just-in-time property.

By Proposition 6.1 and using the same line of argument presented in Proposition 6.3, we can conclude that any attack that exists upon the protocol model that supplies A with an infinite number of fresh values of type T (reflecting any degree of parallelism of A) and allows the intruder to store any number of them, can be mapped to an equivalent attack upon the reduced version of the system, where the intruder is bounded by condition 3 above.

■

Generalising Proposition 6.4 to Proposition 6.5 is quite straightforward, because the just-in-time property is maintained by the internal agent A, even though other protocol messages are allowed to contain multiple fresh values of type T. However, when this property is no longer satisfied, the problem of calculating bounds upon the intruder becomes much more complex. This is due to subtle issues arising, such as dependencies arising between the fresh values being introduced. Extending this work, to handle a broader class of protocols, is an area we are interested in pursuing in future research. In the remaining part of this chapter, we highlight some of the main issues and difficulties that would need to be addressed.

Generalising these results can broadly be divided into two independent problems. The first is concerned with handling internal agents that introduce multiple fresh values in a single message. The second concerns the issue of internal agents generating fresh values that rely on the receipt of previously introduced ones. We consider each of these in turn.

In Propositions 6.3 and 6.4, messages could contain at most one value of the type for which fresh values were being introduced on behalf of internal agents. We weakened these conditions in Propositions 6.5 to allow protocol messages to contain multiple values of such a type, apart from those generated on behalf of an internal agent. Restricting the internal agents in this way satisfied the just-in-time property and therefore allowed us to derive bounds upon the intruder accordingly. To weaken these conditions further by allowing internal agents to introduce multiple fresh values in a single message, does not give rise to these bounds as one might expect. The following example illustrates the problem.

Example 6.9 *Consider a protocol with the following first four messages:*

$$\begin{array}{llll}
\textit{Message } 1. & A \to B & : & \{k_1,\ k_2,\ A\}_{PK(B)} \\
\textit{Message } 2. & A \to S & : & \{B,\ n_a\}_{SKey(A)} \\
\textit{Message } 3. & S \to B & : & \{k_2,\ k_3,\ n_a\}_{SK(S)} \\
\textit{Message } 4. & S \to A & : & \{k_4,\ k_5,\ n_a\}_{SK(S)} \\
 & \vdots & &
\end{array}$$

where k_1, k_2, k_3 and k_4 are keys introduced freshly by the agent A and the server S respectively, and n_a is a fresh nonce introduced by A. Note that the encryption in message 3 is with S's secret key, denoting S's signature.

Suppose S is modelled as internal; its functionality is captured by two generation sets, namely one for the generation of message 3's and one for the generation of message 4's, each introducing two fresh keys.

Consider the following valid trace of events, where Alice and Bob take on the roles of A and B respectively:

(1) *Message* 1. $Alice \rightarrow I_{Bob} : \{K_1, K_2, Alice\}_{PK(Bob)}$

(2) *Message* 2. $Alice \rightarrow I_{Server} : \{Bob, N_1\}_{SKey(Alice)}$

(3) *The intruder generates a message 3 on behalf of S:*

$$\langle K_3, K_4 \rangle,\ \{Bob,\ N_1\}_{SKey(Alice)} \ \vdash \ \{K_3,\ K_4,\ N_1\}_{SK(Server)}$$

(4) *The intruder generates a message 4 on behalf of S:*

$$\langle K_5, K_6 \rangle,\ \{Bob,\ N_1\}_{SKey(Alice)} \ \vdash \ \{K_5,\ K_6,\ N_1\}_{SK(Server)}$$

(5) *The intruder deduces $\{K_3,\ K_5,\ Alice\}_{PK(Bob)}$ using the messages generated in* (3) *and* (4).

(6) *Message* 1. $I_{Alice} \rightarrow Bob : \{K_3, K_5, Alice\}_{PK(Bob)}$

(7) *Message* 3. $I_{Server} \rightarrow Bob : \{K_3, K_4, N_1\}_{SK(Server)}$

(8) *Message* 4. $I_{Server} \rightarrow Alice : \{K_5, K_6, N_1\}_{SK(Server)}$

To construct this trace, the intruder must be able to store 4 fresh keys generated on behalf of S and unknown to any external agents. This is because he gathers the first two keys (K_3 and K_5) of the generated messages 3 and 4, and plays them as a message 1 to Bob. The keys K_4 and K_6 generated along side of them have to be fresh, as they are subsequently passed on to Bob and Alice respectively.

■

If we applied the bounds derived in Proposition 6.5 to the messages generated on behalf of the internal server S above, then the bound would have been 2 and the intruder would not have been able to perform this trace.

The problem with allowing an internal agent to introduce multiple fresh values in a single message, is that dependencies between these values arise. Values that are supplied freshly within a single generation and subsequently all received by external agents will only satisfy the just-in-time property if all of them are passed on at the same time (namely, just-in-time!). However, it is possible (and frequent) that the intruder wishes to break these values up into different messages, pass a subset of them on to an external agent immediately, whilst storing others for later use. Rearranging messages in this way could be crucial ammunition for attacking the protocol. The fact that he uses all of them eventually, means that none of them can be substituted by dummy values.

In order to derive suitable bounds upon the intruder, one would need to take a number of additional factors into account, such as the number of values the intruder can request in a single generation, how these values can be distributed to external agents and what combinatorics the intruder is able to take advantage of. One could consider a number of different classes of protocols that introduce multiple fresh values by internal agents, but are still subject to certain conditions. For example, allowing an internal agent to introduce fresh values within a single message M, on the condition that M is encrypted under a key assumed by the model to be secure (as is frequently assumed with long term keys shared between a server and agents). In this case, one could possibly take advantage of the assumption that the intruder could not split any of these messages up.

The second issue that needs to be considered when generalising these results, is the case where internal agents rely on receiving values that they previously introduced as fresh, in order to generate further fresh values. This is a more subtle problem than the one concerned with internal agents introducing multiple fresh values, and causes complications even when all the protocol messages contain single values of the corresponding types in the system. We illustrate the problem with the following simple example.

Example 6.10 *Consider a protocol with the following first four messages:*

$$\begin{array}{lll} \textit{Message } 1. & S \rightarrow A & : \{k_1\}_{SKey(A)} \\ \textit{Message } 2. & A \rightarrow S & : \{k_1,\ n_a\}_{SKey(A)} \\ \textit{Message } 3. & S \rightarrow A & : \{k_2,\ n_a\}_{SKey(A)} \\ \textit{Message } 4. & S \rightarrow A & : \{k_3,\ n_a\}_{SKey(A)} \end{array}$$

where k_1, k_2 and k_3 are keys freshly introduced by the server S, and n_a is a nonce freshly introduced by the agent A.

Suppose S was modelled as internal. Given an infinite supply of fresh keys, the intruder could, for example, perform a run on behalf of S and generate a message 3 and 4 without passing either of them on to external agents.

He could then send them in reverse order; if S was modelled as an external agent process, this would simply be achieved by the intruder intercepting these messages from the network and then sending them in reverse order. If we are not careful and, for example, place a bound of 1 upon the intruder, then he would be unreasonably restricted from performing this legitimate behaviour.

■

To handle this type of behaviour, a new mapping strategy is needed to reduce a system with an infinite supply of fresh values to an equivalent finite version. The just-in-time property we previously used is no longer applicable here (as illustrated in Example 6.10), since it relies on the fact that (i) no dependencies arise between fresh values introduced by internal agents and (ii) internal agents are no able to distinguish between protocol runs.

When constructing such a mapping, the idea of mapping all internal fresh values to dummy ones is still applicable. To recall, internal fresh values are those generated by the intruder on behalf of internal agents, but are not subsequently passed on to external agents; they are used for internal manipulations within the intruder only. The questions still to be considered are: How do we construct a reduced model with appropriate bounds upon the intruder that is equivalent to the infinite version? For a given protocol (of the form presented in Example 6.10), what factors affect the maximum number of fresh values the intruder needs to store, at any one point, in order to encompass his full range of abilities? Given these factors, how can we make use of them to derive the necessary upper bounds upon the intruder?

A plausible avenue is to extend our notion of just-in-time to one that takes other factors into account, such as the context or run that a fresh value is being generated in. Consider the specific behavioural pattern displayed in Example 6.10. For any run R, performed on behalf of the internal server S, the intruder must be able to store enough fresh values generated on behalf of S, to allow him to pass them on to external agents in any order; in this example, this would be 2.

The new notion of just-in-time would need to be defined such that it is satisfied by traces such as

- Intruder generates message 1 (on behalf of S): $\{K_1\}_{SKey(Alice)}$
- Message 1. $I_{Server} \rightarrow Alice : \{K_2\}_{SKey(Alice)}$
- Message 2. $Alice \rightarrow I_{Server} : \{K_3,\ N_A\}_{SKey(Alice)}$
- Intruder generates message 3 (on behalf of S): $\{K_2,\ N_A\}_{SKey(Alice)}$
- Intruder generates message 4 (on behalf of S): $\{K_3,\ N_A\}_{SKey(Alice)}$
- Message 3. $I_{Server} \rightarrow Alice : \{K_3,\ N_A\}_{SKey(Alice)}$
- Message 4. $I_{Server} \rightarrow Alice : \{K_2,\ N_A\}_{SKey(Alice)}$

of the system described in Example 6.10. In our current definition of just-in-time, the generation of K_2 does not satisfy the property. It seems that the point at which an internal agent is able to identify a particular run, may be a useful factor to consider; in Example 6.10, this would be linked to the receipt of message 2 by the internal server, containing the fresh key previously introduced.

6.5 Example: Capturing parallel attacks

This example demonstrates how the work presented in this chapter concerning internalising agents and the use of dummy values are effective in practice. Although this protocol is rather contrived for the purposes of illustration, it clearly shows how important and effective the work is regarding internal agents. The attack found upon this protocol requires 3 instances of an initiator agent A running in parallel, together with 1 instance of a responder agent B. Furthermore, each instance of A and B requires 3 fresh nonces and 1 fresh nonce respectively. In the standard CSP model generated by Casper without the use of data independence, we would need a minimum of 10 fresh nonces to capture the attack. This is not a feasible model to run through FDR using our standard model checking techniques. This type of *parallel* attack presented here is significant because it is well known that they cause the state space to explode. However, as we demonstrate below, if we model A as an internal agent and declare one instance of B as an external process, we are able to find the attack using 4 fresh values only.

6.5.1 Protocol description

This protocol example was derived from a simpler example known as the *ffgg* protocol presented in [Mil99] and defined as follows[2]:

$$
\begin{array}{lll}
\text{Message 1.} & A \rightarrow B & : \; n_1,\ n_2,\ n_3 \\
\text{Message 2.} & B \rightarrow A & : \; \{n_1,\ n_2 \ \%\ x,\ n_3 \ \%\ y\}_{PK(A)} \\
\text{Message 3.} & B \rightarrow A & : \; \{n_1,\ n_2,\ n_3,\ m\}_{PK(A)} \\
\text{Message 4.} & A \rightarrow B & : \; \{x,\ y,\ m\}_{PK(A)} \ \%\ v \\
\text{Message 5.} & A \rightarrow B & : \; x
\end{array}
$$

where A and B are agents, n_1, n_2 and n_3 are nonces and m, is also a nonce that is introduced by B and intended to remain secret. We use the standard %-notation of Casper; for example, B is prepared to receive a message 4 of the form $\{x,\ y,\ m\}_{PK(A)}$ for any values bound to x, y and m. Note that this message 4 is encrypted under A's public key, so B will not actually be able to decrypt it.

[2] Although intentional, the structure of message 6 is rather odd since B is willing to accept any message encrypted under A's public key. However, as stated in [Mil99], we do not claim any practical use for this particular protocol; rather it provides a useful example to illustrate the advantage of internal agents when it comes to capturing parallel attacks.

The specification we are interested in here is $Secret(B,\ m,\ [A])$; namely, the secret nonce m, introduced by B, is only ever known by B and A. Assuming that *Alice* and *Bob* play the roles of A and B respectively, the parallel attack upon this specification is as follows:

$$
\begin{array}{lrl}
\text{Message } 1.(\alpha) & Alice_1 \rightarrow Bob & : N_1,\ N_2,\ N_3 \\
\text{Message } 2.(\alpha) & Bob \rightarrow I_{Alice} & : \{N_1,\ N_2,\ N_3\}_{PK(Alice)} \\
\text{Message } 1.(\beta) & Alice_2 \rightarrow I_{Bob} & : N_4,\ N_5,\ N_6 \\
\text{Message } 1.(\gamma) & Alice_3 \rightarrow I_{Bob} & : N_7,\ N_8,\ N_9 \\
\text{Message } 2.(\alpha) & I_{Bob} \rightarrow Alice_1 & : \{N_1,\ N_4,\ N_7\}_{PK(Alice)} \\
\text{Message } 3.(\alpha) & Bob \rightarrow Alice_1 & : \{N_1,\ N_2,\ N_3,\ M\}_{PK(Alice)} \\
\text{Message } 4.(\alpha) & Alice_1 \rightarrow I_{Bob} & : \{N_4,\ N_7,\ M\}_{PK(Alice)} \\
\text{Message } 2.(\beta) & I_{Bob} \rightarrow Alice_2 & : \{N_4,\ N_7,\ M\}_{PK(Alice)} \\
\text{Message } 3.(\beta) & I_{Bob} \rightarrow Alice_2 & : \{N_4,\ N_5,\ N_6,\ N_6\}_{PK(Alice)} \\
\text{Message } 4.(\beta) & Alice_2 \rightarrow I_{Bob} & : \{N_7,\ M,\ N_6\}_{PK(Alice)} \\
\text{Message } 2.(\gamma) & I_{Bob} \rightarrow Alice_3 & : \{N_7,\ M,\ N_6\}_{PK(Alice)} \\
\text{Message } 3.(\gamma) & I_{Bob} \rightarrow Alice_3 & : \{N_7,\ N_8,\ N_9,\ N_9\}_{PK(Alice)} \\
\text{Message } 4.(\gamma) & Alice_3 \rightarrow Bob & : \{M,\ N_6,\ N_9\}_{PK(Alice)} \\
\text{Message } 5.(\gamma) & Alice_3 \rightarrow I_{Bob} & : M \\
\text{Message } 5.(\gamma) & I_{Alice_3} \rightarrow Bob & : M
\end{array}
$$

where α, β and γ highlight the three independent runs taking place, and $Alice_i$ represents the i^{th} instance of *Alice* as A.

On receiving Message 5, *Bob* believes that M is a secret shared only with *Alice*. However, through the sending of Message 5 the intruder learns M.

This attack is essentially based on the fact that when *Alice* receives the messages 2 and 3, she sends back a corresponding message 4, which has the effect of permuting the last two values in message 2 and the last value in message 3. The interactions with $Alice_1$ has the resulting effect of incorporating the secret nonce M into this permuting sequence. Once M is part of this sequence, it is just a case of having enough instances of *Alice* to permute this sequence until M is at the head. In this example, this requires two further permutations, after which $Alice_3$ then announces the head of this sequence un-encrypted in message 5. The intruder takes advantage of the fact that messages 2 and 4 have the same format, thereby allowing him to replay message 4 as message 2. He also relies on the fact that *Bob* places no demands on which values he receives in message 4.

Although this attack looks feasible to find in hindsight, it turns out that this model, consisting of 3 initiator agents and 1 responder agent running in parallel, requiring a total of 10 fresh nonces, is not feasible to run through FDR. The state space explodes tremendously due to the parallel runs and the combinatorics of each agent, given the high number of fresh nonces in the network. In fact, we derived this attack from the equivalent attack, presented in the following section, that we found using the data independence techniques, internalisation of agents and the use of dummy values.

6.5.2 The resulting CSP protocol model

We now present the CSP model for this protocol, where we make full use of the data independence techniques together with the work we have been presenting thus far in this thesis concerning the internalisation of agents.

The CSP model is designed as follows. We declare one instance of B is modelled as an external agent process, with identity *Bob*, and model A as an internal agent. We check for the same secrecy specification between *Bob* and *Alice* playing A and B respectively. Therefore the only signal events we require are those associated with *Bob*; these are constructed through the renaming of *Bob's* receive events of message 5, in the standard way.

Bounding the intruder's appetite for fresh nonces

By modelling A as internal, our model does not satisfy the conditions in Proposition 6.3 or Proposition 6.4, since A introduces 3 fresh values in message 1, through the following generation:

$$\langle n_1,\ n_2,\ n_3\rangle,\ \{\} \ \vdash \ n_1,\ n_2,\ n_3$$

Clearly, it would be incorrect to restrict the intruder such that he can only store one fresh nonce (unknown to any external agents) at a time, since this would prevent him from ever being able to generate a message 1 on behalf of A. As we discuss in Section 6.6, deriving suitable bounds upon the intruder for a more general class of protocols such as this one, is very difficult. For the purposes of this example, we place the intruder under the most stringent bounds possible that still allows the attack to be found. We allow him to store at most 3 fresh nonces, unknown to any external agents, at any time, and give the Nonce Manager with 4 fresh nonces to supply the network with.

Thus, if the intruder has reached the limit of the number of fresh nonces he can store, or the Nonce Manager has run out of fresh nonces for the time being, then the intruder can still generate message 1's on behalf of A using the dummy nonce N_D:

$$\langle N_D,\ N_D,\ N_D\rangle,\ \{\} \quad \vdash \quad N_D,\ N_D,\ N_D$$

Even under these stringent conditions, this model provides a good example of how well parallel attacks are captured using the internal agent design.

Capturing the parallel attack

We find the same secrecy attack presented above, as follows:

1. *(Generation 1)* The intruder performs the functionality of A to generate the message 1: $\langle N_1,\ N_2,\ N_3\rangle$, receiving the fresh values N_1, N_2 and N_3 from the Nonce Manager.

2. The intruder then sends this message 1 to *Bob* on behalf of *Alice*:

   $$\text{Message 1.} \quad I_{Alice} \rightarrow Bob \ :\ N_1,\ N_2,\ N_3$$

3. *Bob* responds by sending the following message 2, intended for *Alice*:

   $$\text{Message 2.} \quad Bob \rightarrow I_{Alice} \ :\ \{N_1,\ N_2,\ N_3\}_{PK(Alice)}$$

4. *Bob* then sends the subsequent message 3, also intended for *Alice*:

   $$\text{Message 3.} \quad Bob \rightarrow I_{Alice} \ :\ \{N_1,\ N_2,\ N_3,\ M\}_{PK(Alice)}$$

5. *(Generation 2)* The intruder performs the functionality of A to generate a second message 1: $\langle N_D,\ N_D,\ N_D\rangle$, using the dummy values only.

6. *(Deduction 1)* The intruder performs the functionality of A (as $Alice_1$) to deduce the corresponding message 4:

   $$\left\{\begin{array}{l} \{N_1,\ N_2,\ N_3,\ M\}_{PK(Alice)}, \\ \{N_1,\ N_D,\ N_D\}_{PK(Alice)}, \\ N_1,\ N_2,\ N_3 \end{array}\right\} \quad \vdash \quad \{N_D,\ N_D,\ M\}_{PK(Alice)}$$

If this instance of A was modelled as external, then this would be captured by the following sequence of steps:

$$\begin{array}{lrl}
\text{Message 2.} & I_{Bob} \rightarrow Alice_1 & : \{N_1,\ N_D,\ N_D\}_{PK(Alice)} \\
\text{Message 3.} & I_{Bob} \rightarrow Alice_1 & : \{N_1,\ N_2,\ N_3,\ M\}_{PK(Alice)} \\
\text{Message 4.} & Alice_1 \rightarrow I_{Bob} & : \{N_D,\ N_D,\ M\}_{PK(Alice)}
\end{array}$$

where the dummy nonces would be replaced by fresh ones.

7. *(Deduction 2)* The intruder performs the functionality of A (as $Alice_2$) to deduce the following message 4:

$$\left\{ \begin{array}{l} N_D,\ \{N_D,\ N_D,\ M\}_{PK(Alice)}, \\ \{N_D,\ N_D,\ N_D,\ N_D\}_{PK(Alice)} \end{array} \right\} \vdash \{N_D,\ M,\ N_D\}_{PK(Alice)}$$

If this instance of A was modelled as external, then this would be captured by the following sequence of steps:

$$\begin{array}{lrl}
\text{Message 2.} & I_{Bob} \rightarrow Alice_2 & : \{N_D,\ N_D,\ M\}_{PK(Alice)} \\
\text{Message 3.} & I_{Bob} \rightarrow Alice_2 & : \{N_D,\ N_D,\ N_D,\ N_D\}_{PK(Alice)} \\
\text{Message 4.} & Alice_2 \rightarrow I_{Bob} & : \{N_D,\ M,\ N_D\}_{PK(Alice)}
\end{array}$$

where the dummy nonces would be replaced by fresh ones.

8. *(Deduction 3)* The intruder performs the functionality of A (as $Alice_3$) to deduce the following message 4:

$$\left\{ \begin{array}{l} N_D,\ \{N_D,\ M,\ N_D\}_{PK(Alice)}, \\ \{N_D,\ N_D,\ N_D,\ N_D\}_{PK(Alice)} \end{array} \right\} \vdash \{M,\ N_D,\ N_D\}_{PK(Alice)}$$

If this instance of A was modelled as external, then this would be captured by the following sequence of steps:

$$\begin{array}{lrl}
\text{Message 2.} & I_{Bob} \rightarrow Alice_3 & : \{N_D,\ M,\ N_D\}_{PK(Alice)} \\
\text{Message 3.} & I_{Bob} \rightarrow Alice_3 & : \{N_D,\ N_D,\ N_D,\ N_D\}_{PK(Alice)} \\
\text{Message 4.} & Alice_3 \rightarrow I_{Bob} & : \{M,\ N_D,\ N_D\}_{PK(Alice)}
\end{array}$$

where N_D represents fresh values that $Alice_3$ would have introduced as an external agent. In this case, the intruder does not need to perform another generation, since he already knows the value N_D (as it is public in message 1).

9. *(Deduction 4)* The intruder performs the functionality of A (as $Alice_3$) to deduce the following message 5:

$$\left\{ \begin{array}{l} \{N_D,\ M,\ N_D\}_{PK(Alice)}, \\ \{M,\ N_D,\ N_D\}_{PK(Alice)}, \\ \{N_D,\ N_D,\ N_D,\ N_D\}_{PK(Alice)},\ N_D \end{array} \right\} \vdash\ M$$

   If this instance of A was modelled as external, then this would be captured by the following sequence of steps:

   Message 5. $Alice_3 \rightarrow I_{Bob}\ :\ M$

   As a result of this message, the intruder now knows the secret nonce M.

10. The intruder sends the following message 4 to *Bob*, pretending to be from *Alice*:

    Message 4. $I_{Alice} \rightarrow Bob\ :\ \{N_1,\ N_2,\ M\}_{PK(Mallory)}$

11. Finally, the intruder sends the following message 5 to *Bob*, pretending to be from *Alice*:

    Message 5. $I_{Alice} \rightarrow Bob\ :\ N_1$

On receiving message 5 in Step 11, *Bob* believes that he has completed the run of the protocol where his secret nonce M has only been shared with *Alice* only. As we see in Step 9, the intruder learns M, by making use of 3 concurrent runs of A, with identity *Alice*. The dummy value N_D in all the generations and deductions represent fresh nonces that would have been introduced, had each instance of the *Initiator* agent involved in the attack been modelled as an external agent process.

6.6 Discussion

Prior to the work presented in this chapter, the data independence techniques allowed us to capture unbounded sequential runs for each participating agent, within the confines of a finite refinement check (using the FDR model checker). Our models were, however, restricted when it came to capturing any degree of parallelism within a single agent identity; the limiting factor was not the modelling techniques themselves, but rather the enormous effect any additional agent instances had upon the state space. Apart from the simplest of protocols, modelling agents with a degree of parallelism of more than one, becomes rapidly infeasible.

In this chapter, we showed how the internalisation strategy (developed in Chapter 5) can overcome this problem, and enable our models to feasibly capture attacks upon protocols that require a higher degree of parallelism than previously possible. We derived a number of classes of protocols (in Propositions 6.3, 6.4 and 6.5) for which we are able to capture attacks that require any degree of parallelism amongst the identities of the agents modelled internally within the intruder. Whilst the conditions required in these classes are quite stringent (mainly upon the internal agents), there are still very useful and applicable to many protocol models. In particular, most server-type roles can be modelled as internal and satisfy the necessary conditions.

This work has enabled us to perform a more complete analysis of security protocols using model checking techniques than was previously possible. The example presented in Section 6.5 illustrates this well. The attack required three instances of the initiator agent (*Alice*), together with a single instance of the responder (*Bob*). Using old modelling techniques, this model is infeasible to run; using our new techniques and internalising the initiator agent, meant that the attack was found very easily.

6.6.1 Further research

The work presented in Chapter 5 regarding how signal events can be constructed for internal agents, strengthens this work even further, since it allows our models to capture parallelism amongst agents that are part of a specification. Further still, it opens potential avenues for capturing more complete results upon protocol analysis, than simply parallelism of internal agents. This involves carefully selecting which agent roles are internalised and which are modelled as external; this is dependent on the specification being verified. In the remaining part of this chapter, we suggest how this could potentially be achieved for secrecy and authentication properties in turn. This work is part of future research we are interested in pursuing.

Secrecy specifications

All secrecy specifications have the form $Spec(A, s, [B_1, \ldots, B_n])$ where *Spec* is either *Secret* or *StrongSecret*, and A is a trustworthy agent who believes that the value bound to variable s is a secret shared only with the agents $B_1, \ldots, B_n$. We require only one type of signal event within our protocol model when verifying these specifications; for *Secret*, it is identified with the last message *received* by A and in the case of *StrongSecret*, it is identified with the first message containing s. Formal definitions of these specifications are given in Chapter 2 on Page 44.

Suppose we are interested in modelling a protocol P with the set of roles *AS*, and verifying secrecy specifications of the form $Spec(A, s, [B_1, \ldots, B_n])$, where $A, B_1, \ldots, B_n$ are members of *AS*. Suppose further that we construct this CSP model as follows:

1. All agents in AS that are modelled as internal.
2. One instance of A is modelled as an external agent process.
3. The necessary signal events (required for A only) are constructed for the external instance of A; none are constructed for any internal instances of A.

We believe that, under appropriate bounds upon the intruder (such as those derived in the earlier propositions), if no secrecy attack found in this model, then none exists upon P for any degree of parallelism amongst *AS*.

We justify this claim as follows. The functionalities of all the agents in AS are internalised within the intruder and therefore, given that we have calculated an appropriate bound upon him, this represents an unbounded degree of parallelism amongst them (for the same reasons described in the earlier propositions). By definition, a secrecy specification of the form $S(A, s, [B_1, \ldots, B_n])$ is broken precisely when there exists an instance of A who believes that the value v bound to s is shared only with $B_1, \ldots, B_n$, while in fact the intruder has been able to deduce v. It does not matter which instance of A this is;

the only requirement is that there exists one of them. Our model has one external instance A_E of A with the appropriate signal event linked to it, and so any attack the intruder can perform upon an internalised instance of A, he can also perform upon A_E. Therefore, it suffices to have signal events for the secrecy specifications linked only to A_E, and none associated to the internal instances of A.

Our current limitation with this result is that we have not derived bounds upon the intruder for all general cases; our current propositions are somewhat limited when internalising multiple agent roles. However, deriving such bounds is a current area of our future research, thereby making a more complete analysis of protocols as presented above, a feasible achievement.

Authentication specifications

We can apply a similar approach for capturing authentication properties for any degree of parallelism within the models. The class of authentication specifications we consider are those that do not demand a one-to-one relationship between the runs of the agents in question (the reason for this becomes apparent later on).

An authentication specification takes the form $Spec(A, B, xs)$, representing the property that A must be authenticated to B and agree on the values bound to the variables in xs. Two types of signal events are required: *Signal.Running* and *Signal.Commit* events. The former of these is identified with the last message sent by A and the latter is bound to the last message participated by B (and transitively linked to A as discussed in Chapter 2).

Suppose we model a protocol P with the set of roles *AS*, and verify authentication specifications of the form $Spec(A, B, [x_1, \ldots, x_n])$, where A and B are members of *AS*. Suppose further that we construct this CSP model as follows:

1. All agents in AS that are modelled as internal.
2. One instance of B is modelled as an external agent process.
3. The *Signal.Commit* events (required for B) are constructed for the external instance of B only; none are constructed for any internal instances of B.
4. The *Signal.Running* events (required for A) are constructed for all internal instances of A.

We believe that, under appropriate bounds upon the intruder (such as those derived in the earlier propositions), if no authentication attack found in this model, then none exists upon P for any degree of parallelism amongst AS.

We justify this claim as follows. The functionalities of all the agents in AS are internalised within the intruder and therefore, given that we have calculated appropriate bounds upon him, this represents an unbounded degree of parallelism amongst them. We know that the specifications in question do not demand a one-to-one relationship between the runs of the two agents being verified. An attack will be found upon these specifications if there exists at least one case where if some instance of B (and we don't care which!) commits himself to a run of the protocol believing he has done so with A, when in fact A has not been running the protocol (according to the definition of one of the three possible specification we are considering). For each specification, it does not matter which instance of A performs the necessary *Signal.Running* as long as there exists one of them that is in that state. Therefore, we can internalise all instances of A within the intruder and capture the *Signal.Running* events that we are interested in through the corresponding deductions and generations. Furthermore, it does not matter which instance of B participates in the run that leads to the authentication attack; any authentication attack the intruder can perform upon an internalised B, he can also perform upon the one external instance B_E of B. It therefore suffices to have *Signal.Commit* events for the linked only to B_E, and none associated to the internal instances of B.

Chapter 7

Discussion

7.1 Conclusion

In this thesis, we have presented an approach to analysing security protocols within the general-purpose model checking world, more complete than previously possible. There are many different approaches used today for the verification of security protocols (summarised in Chapter 1) and we discuss how our work compares to them later on. Developing such methods successfully has proved to be a difficult task and most of them suffer from one or more of the following problems: Lack of tool support, state space explosion problems, inefficiency, incomplete or restricted analysis (either due to models being restricted to a small finite size or non-terminating), lack of expressive power, high degrees of expertise requirements and ambiguity. This list is by no means exhaustive. Modelling techniques created to succeed strongly in a subset of these aspects often tend to fall short in others. The same applies to model checking techniques.

The advantage of model checking techniques is that the analysis phase of a protocol is fully automatic and requires little or no guidance from an expert user. Although expertise is required to construct the protocol model, it is becoming more common for designers to implement compilers that translate a high level description of a security protocol (in the form commonly found in the literature) to the input language of a given model checker. For example, throughout this thesis we have been using and extending the compiler Casper (by Lowe [Low98]) that generates the corresponding CSP models for the FDR model checker (described in more detail in Chapter 2). Others include CAPSL [Mil97], a Common Authentication Protocol Specification Language that translates high level descriptions into a number of existing tools (for example, the NRL Protocol Analyser [BMM99] and Athena [Mil00]), and CVS [DFG99] that translates high level descriptions into the SPA language (Security Process Algebra).

The disadvantage of model checkers is that the state space being searched needs to be finite (in order to terminate) and therefore their use for *proving* the correctness of protocols is limited. Furthermore, state space explosion is a well known phenomenon and although many reduction strategies have been developed over the years, it still poses problems when it comes to the feasibility of analysing larger protocol models.

The aim of this thesis was to enhance our model checking framework with the data independence techniques developed in [Ros98a, RB99], and enable a more complete analysis of security protocols than previously possible. The results achieved are summarised below.

The implementation and automation of data independence. In Chapter 4, we presented the design and implementation of the data independence techniques presented in Chapter 3. These techniques allow us to model unbounded sequential runs for the participating agents, within the confines of a finite refinement check, thereby enabling a deeper analysis of protocols than hitherto. Constructing such models is time-consuming, error-prone and requires a substantial knowledge of the CSP language as well as the theory underlying the data independence techniques. To overcome this and make our techniques accessible to a broader class of users, we automated them within the Casper compiler. As illustrated in the example scripts presented in Chapter 4, the complexity of the design and implementation are abstracted away from the users of Casper.

State space reduction strategies. We developed three effective state space reduction strategies, making our modelling techniques much more efficient and enabling us to capture larger and more complex protocol models. The first (presented in Chapter 4) involved optimising the basic design of the CSP models generated by Casper, with regards to the construction of the signal events. Rather than explicitly including them in the definition of the participating agents, we constructed them through appropriate renamings of the corresponding send and receive events. This reduced the state space by one order of magnitude. The second reduction strategy (presented in Chapter 4) was developed specifically for the data independence models, exploiting the recycling mechanism. This involved supplying the intruder's initial state with all stale knowledge (messages containing stale values after they have been recycled) that he is able to gain eventually. The impact upon the state space is an improvement of two orders of magnitude. The third state space reduction strategy (presented in Chapter 5) involved introducing a new approach for modelling protocol roles, known as internal agents. This has improved our modelling techniques by a further two orders of magnitude. Thus, combining these techniques addresses the state space explosion problem very effectively.

Capturing parallelism within agents. A drawback of many model checking techniques (including our own) is that we are restricted to modelling small finite systems (as discussed above). We overcame this partially with the data independence work presented in Chapters 3 and 4, where we enabled our models to capture unbounded sequential runs for each participating agent. The problem of capturing a higher (and even unbounded) degree of parallelism remained infeasible. In Chapter 6, we showed how the internalisation strategy (developed in Chapter 5) can be used to enhance our modelling techniques further and enable our models to feasibly capture attacks upon protocols that require a higher degree of parallelism than previously possible. Furthermore, we derived a number of classes of protocols (in Propositions 6.3, 6.4 and 6.5) for which we are able to capture attacks that require any degree of parallelism amongst the identities of the agents modelled internally within the intruder. The work presented in Chapter 5 regarding how signal events can be constructed for internal agents, strengthens this work even further, since it allows our models to capture parallelism amongst agents that are part of a specification.

7.2 Comparison with other existing approaches

The data independence work presented in this thesis has contributed towards a substantial improvement over other similar model checking techniques such as Murϕ (by Mitchell *et al.* [MMS97]), Brutus (by Marrero *et al.* [MCJ97]) and CoSec (by Focardi and Gorrieri [FG97]). Even though these methods (like ours) are very useful for finding attacks (upon small instances of protocols), they still suffer from the problem of only being able to model a small finite number of agents, each performing a small number of finite runs. Our work has moved the CSP and FDR framework from this class of tools towards one that can handle and reason about protocols of a much larger and in some cases arbitrary size, without losing the benefits that the model checking approach provides.

Many of the tools have been purpose-built for the analysis of security protocols. Some suggest that this is advantageous with regards to efficiency and general usability, since they are developed and fine tuned solely for this problem domain. For example, Buttyán [But99] makes the following sweeping statement: *"Although, there are notable attempts to use general purpose verification tools for analysing cryptographic protocols, they are, in general, not suitable for this purpose. The main weakness of this approach is that in applying methods that were not intended specifically for security analysis, subtle pitfalls that are peculiar to the security domain are not considered..."*. This view seems very ill-informed and unreasonably critical to a developing area of modelling that has proved to be successful on numerous occasions. The shortfalls mentioned are not necessarily a direct result of using general purpose tools, but rather a reflection of their misuse.

There are also advantages to be gained by using an already established and rich framework like CSP; for example, CSP is an expressive language designed primarily to model communication between processes, as well as having a developed underlying mathematical theory to reason about such systems and extensive tool support. There are a number of approaches that are restricted to modelling safety properties and are not powerful enough, for example, to reason about liveness. In this thesis, we used the traces model since we were only concerned with safety properties. However, the CSP framework provides two richer models, namely the stable failures and the failures/divergences models, that enable us to capture properties such as liveness. This opens the way for extending our security protocol models further to consider properties such as denial of service. We discuss this in more detail further on as an avenue for future work. It seems that there are advantages to be gained from both approaches, as Meadows ably puts it in [Mea96a]: *"some of the advantages of using an existing system is its greater power and maturity, while the advantages of using one's own is that it can be tailored to the problem at hand, giving better assistance for solving problems of interest"*.

On the other side of the spectrum from model checking lie theorem proving techniques. These tend to have the advantage of being able to reason about arbitrarily sized models with relative ease (unlike most model checking approaches). The inductive approach by Paulson [Pau98] is a well known example that falls into this category, using the theorem prover Isabelle. This has proved to be very effective for proving correctness of protocols, but does require a high level of interaction and expertise to generate them. Automating such methods is a difficult task. Another approach is the spi-calculus of Abadi and Gordon [AG99]. This is an extension of the π-calculus (with cryptographic primitives) and provides an elegant framework for analysing security protocols. The lack of tool support does, however, limit their accessibility to a broader class of users. One of the aims of our work was to ensure that the data independence ideas used did not weaken the automation and accessibility that the CSP and FDR framework (together with Casper) have always provided.

A number of special, purpose-built tools have been developed over the years with the aim of reaping the benefits of both the model checking and the theorem proving approaches: automation and completeness of proofs. A well known example is the NRL Protocol Analyzer by Meadows [Mea96b]. This tool is able to handle arbitrary-sized protocol models using partially automated theorem proving techniques. The problem of state space explosion does arise, although a number of reduction techniques are used to control this. Meadows [Mea96a] did a comparative study of the NRL Protocol Analyzer and our CSP modelling techniques using FDR; the protocol analysed was the Needham-Schroeder Public-Key protocol (original [NS78] and corrected version [Low96a]). One of the main conclusions was that the NRL Protocol Analyzer was more effective when it came to proving the correctness of the corrected version for arbitrary size. On the other hand, FDR was more efficient at searching through the state space and finding the attacks upon the original version, and did not require any user guidance. It seems that model checking and theorem proving based approaches are in many ways complementary. Another method that is complementary to ours is Schneider's approach [Sch98] using rank functions within the CSP framework and the theorem prover PVS. Initially, the analysis was restricted to modelling finite size protocols only; this has been extended by Heather and Schneider [HS00] to allow more complete proofs for some classes of protocols of arbitrary size. Song *et al.* [SBP01] developed the tool Athena, based on the strand spaces work by Fábrega *et al.* [FHG98a, FHG98b]. An advantage Athena has over many is its ability to avoid state space explosion.

Finally, it is worth mentioning a recent approach by Blanchet [Bla01], since some of the ideas used are similar to ours, especially with regards to the internalisation strategy of agents (referring to the early stages of this development in [BLR00]). The author presents a prolog based framework with the following two abstractions: (i) fresh values are represented as functions over the possible pairs of participants and (ii) a protocol step can be executed several times, instead of only once per session. With these abstractions, the author is able to capture unbounded agent runs and degrees of parallelism within their identities. Similar to ours, his analysis is fail-safe in the sense that no attacks are lost; however, the potential for false attacks does arise. Although this formalism does provide a simpler and faster analysis, it does suffer from a number of drawbacks. The first concerns the issue of freshness. By modelling values that are expected to be fresh for every run, as functions over the participants, it is no longer possible to distinguish between old values used in previous ones and new; therefore one cannot verify whether a protocol satisfies this basic requirement. Secondly, this framework is currently only able to capture secrecy properties; this is one of the reasons why the abstraction of the freshness of values is currently plausible (and not a cause for severe false attacks). However, this will be less straightforward when extending the model to handle authentication.

7.3 Future Work

We suggest a number of interesting avenues for future research that builds upon the work presented in this thesis.

Generalising results in Chapter 6. In Chapter 6, we derived a number of simple classes of protocols for which we can model an arbitrary degree of parallelism within the internal agents. The conditions are quite stringent, particularly upon the internal agents' usage of fresh values they introduce. We are interested in extending these results to handle more general classes of protocols, as discussed in more detail in Section 6.6.

Incorporation of time. Thus far, we have not dealt with time in this thesis. The standard Casper models that do not use the data independence techniques do allow time stamps within the models and timed authentication specifications. We would like to extend our data independence work to include time properties.

Scalability to more complex protocols. The main source of examples we have considered whilst developing our data independence techniques, have been the standard ones presented in the literature. Scalability is often limited in model checking techniques (including our own) due to the state space explosion problem. With the techniques we have developed, in particular the state space reduction strategies in Chapters 4 and 5 and the internal agent model, it would be interesting to re-evaluate how scalable our framework has become.

Liveness properties. In this thesis, we considered two safety properties only; namely confidentiality and authentication. There are also important liveness properties that could benefit from formal analysis as they increasingly become a concern in the electronic world of today. Denial of service is a well-known example, where a malicious intruder is able to disrupt or completely break down services provided to legitimate users, networks and/or any other possible resources. As quoted from [MSK99]: *"These denial of service (DoS) attacks cost businesses millions of dollars each year and are a serious threat to any system and network. These costs are related to system downtime, lost revenues, and the physical labor involved in identifying and reacting to such systems."* One of the most famous DoS attacks occurred in September 1996, according to [MSK99]: *"An internet service provider (ISP) out of New York, Public Access Networks Corporations (PANIX), was under seige for over one week, denying Internet service to about 6,000 individuals and 1,000 companies, according to PC Week."* To make matters worse, the attack was very simple and exploited the way in which the TCP/IP protocol (underlying the Internet) handled SYN requests; this has become a known classic denial of service attack. Research in this area is growing, although most tools developed for the verification of security protocols still focus solely on the common safety properties. This would be an interesting avenue to pursue within the CSP/FDR framework, especially with the data independence developments we have made, enabling agents to perform unbounded sequential runs; a model where agents were restricted to only performing one or two runs would be rather limited to denial of service analysis. Denial of service might be related to CSP concepts of deadlock and (more likely) divergence. The ability to model time in the CSP models will also be an advantage when analysing the sort of problem.

Others. The analysis of cryptography and its use in protocols is a growing field and there are many avenues emerging for future research. For example, Meadows [Mea00] presents a number of new problem areas of interest such as open-ended protocols (data fields may be of arbitrary length), anonymous communication and composability (multiple security protocols running in the same environment).

Bibliography

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 149(1):1–70, January 1999.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proceedings of the Royal Society of London*, volume 426, pages 233–271, 1989.

[BAN90] M. Burrows, M. Abadi, and R. Needham. Rejoinder to Nessett. *Operating Systems Review*, 24(2):39–40, April 1990.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society Press, 2001.

[BLR00] P.J. Broadfoot, G. Lowe, and A.W. Roscoe. Automating data independence. In *Computer Security - ESORICS 2000*, volume 1895 of *LNCS*, pages 175–190. Springer, 2000.

[BMM99] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, 1999.

[But99] L. Buttyán. Formal methods in the design of cryptographic protocols (state of the art). Technical Report SSC/1999/38, Swiss Federal Institute of Technology (EPFL), Lausanne, 1999.

[CJ96] J. Clark and J. Jacob. Attacking authentication protocols. *High Integrity Systems*, 1(5):465–474, 1996.

[DFG99] A. Durante, R. Focardi, and R. Gorrieri. CVS : A compiler for the analysis of cryptographic protocols. In *12th IEEE Computer*

Security Foundations Workshop, pages 203–212. IEEE Computer Society Press, June 1999.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22, 1976.

[DS81] D.E. Denning and G.M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[DvOW92] W. Diffie, P.C. van Oorschot, and M.J. Weiner. Authentication and key exchanges. *Design, Codes and Cryptography*, 2:107–125, 1992.

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.

[ES95] E.A. Emerson and A.P. Sistla. Utilizing symmetry when model checking under fairness assumptions: an automata-theoretic approach. In *Proceedings of the 7th CAV*, volume 939 of *LNCS*, pages 309–324. Springer, 1995.

[ES96] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[FG97] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9), 1997.

[FHG98a] F.J.T. Fábrega, J.C. Herzog, and J.D. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[FHG98b] F.J.T. Fábrega, J.C. Herzog, and J.D. Guttman. Strand Spaces: Why is a security protocol correct? In *Proceedings of IEEE Symposium on Security and Privacy*, pages 160–171, 1998.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 Manual*, 1997.

[GNY90] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. *IEEE Computer Society Symposium in Security and Privacy*, pages 234–248, May 1990.

[HL01] M. Hui and G. Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Science*, 2001.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HS00] J.A. Heather and S.A. Schneider. Towards automatic verification of security protocols on an unbounded network. In *13th IEEE Computer Security Foundations Workshop*, pages 132–143. IEEE Computer Society Press, July 2000.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[Laz99] R.S. Lazić. A semantic study of data-independence with applications to the mechanical verification of concurrent systems. D.Phil. thesis, University of Oxford, 1999.

[LB00] G. Lowe and P.J. Broadfoot. *Casper: A Compiler for the Analysis of Security Protocols*, December 2000. Casper manual (version 1.3).

[Low95] G. Lowe. An attack on the Needham-Schroeder Public-Key authentication protocol. In *Information Processing Letters*, volume 56, pages 131–133, 1995.

[Low96a] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key protocol using FDR. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

[Low96b] G. Lowe. Some new attacks upon security protocols. In *9th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1996.

[Low98] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[Low99] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7:89–146, 1999.

[LR97a] R.S. Lazić and A.W. Roscoe. Using logical relations for automated verification of data-independent CSP. In *Proceedings of the Workshop on Automated Formal Methods*, volume 5 of *Electronic Notes in Theoretical Computer Science*, 1997.

[LR97b] G. Lowe and A.W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE transactions on Software Engineering*, 23(10):659–669, 1997.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[Mea91] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195. IEEE Computer Society Press, 1991.

[Mea92] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1:5–53, 1992.

[Mea96a] C. Meadows. Analyzing the Needham-Schroeder Public-Key protocol: A comparison of two approaches. In *European Symposium on Research in Computer Security*. LNCS, Springer-Verlag, 1996.

[Mea96b] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[Mea00] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of DISCEX 2000*, pages 237–250. IEEE Computer Society Press, 2000.

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[Mil99] J. Millen. A necessarily parallel attack. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[Mil00] J. Millen. A CAPSL Connector to Athena. In H. Veith, N. Heintze, and E. Clarke, editors, *Workshop of Formal Methods and Computer Security*. CAV, 2000.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[MSK99] S. McClure, J. Scambray, and G. Kurtz. *Hacking exposed: Network security secrets and solutions*. Osborne/McGraw-Hill, 1999.

[Nes90] D.M. Nessett. A critique of the Burrows, Abadi, and Needham logic. *Operating Systems Review*, 24(2):35–38, April 1990.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[Pau98] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6, 1998.

[RB99] A.W. Roscoe and P.J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security: Special Issue CSFW11*, 7(2,3):147–190, 1999.

[RG97] A.W. Roscoe and M.H. Goldsmith. The perfect 'spy' for model-checking crypto-protocols. In *Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols*, 1997.

[Ros96] A.W. Roscoe. Intensional specifications of security protocols. In *9th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1996.

[Ros98a] A.W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[Ros98b] A.W. Roscoe. *The Theory and Practice of Concurrency*. Computer Science. Prentice Hall, 1998.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[RSG+01] P.Y.A. Ryan, S.A. Schneider, M.H. Goldsmith, G. Lowe, and A.W. Roscoe. *Modelling and analysis of security protocols*. Addison Wesley, 2001.

[SBP01] D.X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security: Special Issue CSFW12*, 9(1,2):47–74, 2001.

[Sch96] B. Schneier. *Applied Cryptography.* John Wiley & Sons, Inc, 1996.

[Sch98] S.A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 1998.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Laboratory, SRI International, 1993.

[SS98] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998.

[Sto99] S.D. Stoller. A reduction for automated verification of authentication protocols. In *Workshop on Formal Methods and Security Protocols*, July 1999.

[SvO94] P. Syverson and P. van Oorschot. On unifying some cryptographic protocol logics. *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28, May 1994.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, July 2001.

[Syv94] P. Syverson. A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 131–136, 1994.

[TMN90] M. Tatebayashi, N. Matsuzaki, and D.B. Newman. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology: Proceedings of Crypto '89*, volume 435 of *LNCS*, pages 324–333. Springer-Verlag, 1990.

[Wel98] D. Welsh. *Codes and Cryptography.* Oxford University Press Inc., 1998.